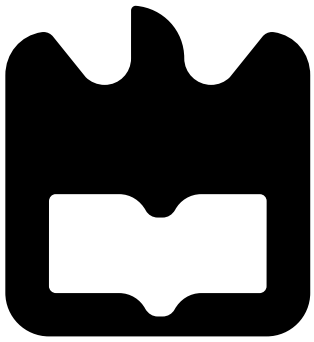
Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática,
2016

Tânia Sofia Vilaranda Alves

Sistema de visão estéreo para estimativa de trajectórias de bolas em 3D

Stereo vision system for 3D ball trajectory estimation



Tânia Sofia Vilaranda Alves

Sistema de visão estéreo para estimativa de trajectórias de bolas em 3D

Stereo vision system for 3D ball trajectory estimation

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requesitos necessários à obtenção do grau de Mestre em Engenharia de Computadores e Telemática, realizada sob a orientação científica de António José Ribeiro Neves e de Paulo Miguel de Jesus Dias, Professores do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

o júri / the jury

presidente / president

Prof. Doutor Armando José Formoso de Pinho
Professor Associado c/ Agregação da Universidade de Aveiro

vogais / examiners committee

Prof. Doutor Jorge Augusto Fernandes Ferreira
Professor Auxiliar da Universidade de Aveiro

Prof. Doutor António José Ribeiro Neves
Professor Auxiliar da Universidade de Aveiro (orientador)

Acknowledgements

To my family who has supported me over my years in college, both financially and emotionally. To my great friends, Vasco Santos, Catarina Bastos and Zé Diogo who have helped me and from whom I have learned alot during these years in college.

To my coordinators, António Neves and Paulo Dias, who have guided me through this thesis and have helped me grow.

To everyone, a big thank you for all you have put up with and for never leaving me hanging.

Resumo

O sistema de visão humano é um sistema complexo e extraordinário. O ser humano usa o que se denomina por esteropsia para obter uma noção da profundidade dos objectos que o rodeiam. Os dois olhos capturam duas imagens do mundo e o cérebro processa e interpreta a informação capturada. Um sistema de visão estéreo é baseado nesta capacidade. O sistema integra duas ou mais camaras que capturam visões diferentes do mundo. As imagens são processadas para obter informação 3D sobre os objectos e do próprio mundo.

Estéreo é uma área importante da visão por computador que é usada em várias aplicações como por exemplo detecção de pessoas, navegação em robôs móveis, automação industrial, veículos de condução autónoma entre outros.

Hoje em dia, a visão por computador começa também a ser usada em desporto, nomeadamente futebol, ténis e *hockey*.

O objectivo desta tese é apresentar um sistema de visão estéreo que consiga detectar objectos coloridos em tempo real e estimar a sua trajectória em 3 dimensões.

Este documento apresenta os passos necessários para obter um sistema funcional incluindo a calibração das camaras, sincronização de camaras, detecção de objectos e a relação da sua posição nas diferentes camaras. Também apresenta experiências e resultados que confirmam a eficácia do sistema.

Abstract

The human vision system is a very complex and amazing system. Humans use what is called stereopsis to perceive depth and 3D objects. The two eyes capture two different images of the surrounding world and it is the brain that processes and interprets the information captured.

A stereo vision system is based on this ability. The system integrates two or more cameras that capture different views of the world. The images are processed to obtain information about 3D objects and the world itself.

Stereo is an important area of computer vision used in several applications such as people tracking, mobile robotics navigation, industrial automation, automonomous vehicle research and others.

Nowadays computer vision is starting to be used in sports as well, namely soccer, tennis and hockey.

The main goal of this thesis is to present a stereo vision system that is able to detect colored objects in real time and estimate their trajectory in 3 dimensions.

This document presents the necessary steps to have a fully functioning stereo system including camera calibration, camera synchronization, object detection and the relation of the its position in the different cameras. It also presents system experiments and results that confirm the effectiveness of the system.

Contents

List of Figures

List of Tables

Acronyms

CAMBADA	Cooperative Autonomous Mobile roBots with Advanced Distributed Architecture
CCD	Charge-Coupled Device
CMOS	Complementary Metal-Oxide Semiconductor
FIFA	Fédération Internationale de Football Association
GND	Ground
GPIO	General Purpose Input/Output
HSV	Hue, Saturation and Value color space
IFAB	International Footbal Association Board
LUT	LookUp Table
PWM	Pulse Width Modulation
RGB	Red Green Blue (colorspace)
RLE	Run-Length Encoding
ROI	Region Of Interest
SDK	Software Development Kit
TCP	Transmission Control Protocol
UEFA	Union of European Football Associations
VTK	Visualization ToolKit
XML	Extensible Markup Language

Chapter 1

Introduction

Humans have the ability to perceive depth and analyze the 3D world by using their two eyes and sending the images to the brain to be processed. The eyes are aligned horizontally and are separated by a small distance, perceiving different images of the same scene in the world. The difference between the images is what enables the brain to extract information about the 3-dimensional objects and the depth of the scene.

A stereo vision system is based on this ability and can be composed of two or more cameras that capture slightly different images of the same portion of the world as seen in figure 1.1.

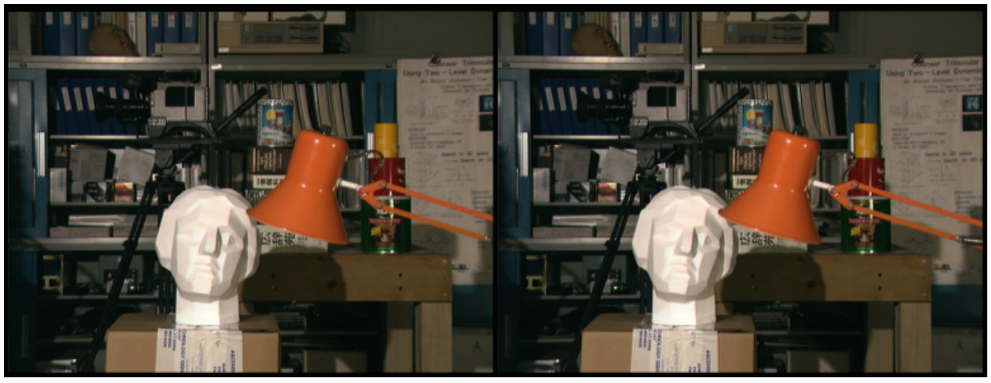


Figure 1.1: The stereo system captures two different views of the scene. On the right there is the right view of the scene and on the left, the left one. These are then processed at the same time to retrieve information about the distance at which the objects are in relation to the camera. [1].

This type of vision system processes the different images and creates a relation between the position of the objects and their real position in the world. The image shown in figure 1.2 is the result of the processing performed on the images present in the figure 1.1.

There has been an increase in research towards vision systems with applications in sports.

Figure 1.2: This image is the result of the processing performed on the previous images where the whiter shades translate to closer objects and the darker shades to further away objects [1].

More recently a few companies have emerged in the field contributing with commercially available systems to major sports associations or companies. This thesis started as a challenge given to the University of Aveiro by a portuguese company, Sports Partner.
Sports Partner is a Portuguese company that is focused on creating solutions for indoor and outdoor sports equipments and floors [2].

To have a fully functional and accurate system there are some steps required before extracting the information from the images. The first one is the calibration of the cameras and the vision system. One possible approach is considering the pinhole camera model that is the simplest and oldest camera model. It is a box shaped camera without a lens and with a small aperture on one of the sides with the size of a pin. The light travels through this hole displaying the image on the opposite side of the box. Modern cameras contain a lens that introduces some characteristics distorting the image captured. For this reason, it is mandatory to calibrate the camera finding the matrices that translate the lens distortion and the alignment between the CCD sensor and the lens.
The second step necessary is to synchronize the cameras to ensure that the changes of the world are captured by both cameras at the same instant in time.

After these steps, the system can now process the images. The object of interest is detected in both captured images and a relation of the position of the objects must be calculated.

1.1 Main Goals

The main goal of this thesis was to create a fully functional stereo vision system that detects an object and estimates its trajectory in real time with the aim to be used in sports like basketball, among others. The steps presented above were implemented and in the end

an algorithm was developed that estimates the trajectory of the object.

For camera calibration, the chessboard method was used to retrieve the radial and tangential distortion matrices. Also the position and translation of each of the integrated cameras must be retrieved.

In order to have the system working in real time, the cameras must be synchonized so that the frames captured correspond to the same instant. The synchronization was implemented using an external trigger that fired the cameras to capture the images at the same time.

The next step was to detect the object in each of the images. Color segmentation followed by blob analysis was the method chosen to obtain the pixel coordinates of the object in the image.

Matrices calculated in the first step were used to convert the pixel coordinates to world coordinates using *OpenCV*. The world coordinates from both cameras are then related to obtain the actual position of the ball in the world and define its trajectory.

The trajectory of the ball is composed by tracing the position of the ball over time and displayed to the user in a 3D VTK scene. The system is also able to calculate the origin and destination mathematically by processing the positions of the trajectory.

Some experiments are also presented in this document confirming the effectiveness of the full system from the camera calibration to the trajectory estimation.

1.2 Document Structure

The structure is as follows:

- Chapter 2 is the state of the art of this thesis. It presents the results of a research for already existing systems with the same or similar goals as this one. It can also be found here information about existing work and/or research where this system was based.

- Chapter 3 explains what is a stereo vision system, what cameras were used in this implementation and the drivers developed to interact with the cameras. The synchronization module that allows the system to estimate the object's trajectory in realtime is also presented in this chapter followed by the camera calibration operations. The chapter is finished by a presentation of the adaptation of the stereo vision system to achieve our goal.

- Chapter 4 describes the overall system. Starting from the way the ball was detected, the calculation of its position on each camera and the relation of the position of the ball in both the cameras. Also, it is presented here how the trajectory is estimated using the coordinates of the ball over time.

- Chapter 5 presents some experiments along with the corresponding results. These are reviewed and conclusions are made for each one.
- Chapter 6 is the conclusion of the work. Here the final considerations on the system are presented alongside the future work that could be performed.

Chapter 2

State of the Art

Nowadays research towards vision systems with applications in sports is increasing with global sports associations or companies looking for goal-decision aiding systems. Lately a few companies have emerged in the field contributing with commercially available systems to these major sports associations. This chapter presents some examples of the commercial system available that either use computer vision or magnetic fields to make the goal decision. Also, it is presented some research towards the same goal, both from inside the University of Aveiro and outside.

2.1 Ball Tracking Systems

Some innovation has been made in this field with the goalline technologies arising. There are several systems approved by either FIFA or IFAB that are either based on analyzing data from computer vision systems or from systems that detect disturbances in magnetic fields.

2.1.1 Computer vision based systems

GoalControl System is composed of 14 cameras (7 on each end side of the field) that cover the whole goal area. Each camera outputs up to 500 frames per second and is connected through optical fiber to the central processing units that are located in the backstage of the stadium. These units are very powerful computers the size of a small closet. Before each game the cameras must be placed in stragetic points and a bundle of calibration operations and accuracy tests must be performed so that the system is validated for that game. Each one of the referees carry a watch connected to the central unit that indicates whether it was goal or not aiding them in the decision making. This is the company that is approved by FIFA and is nowadays used in major soccer games. This system also allows for replays of the goals after the games and is able to recreate them

in a 4D renderized video [3]. Figure 2.1 shows the configuration of the cameras placed in the field (top image) and the smartwatch used by the referees to check the decision of the system on whether it was goal or not (bottom image).

Figure 2.1: The system implements fourteen cameras on the field, seven on each side of the goals. The referees get the information about the score on a smartwatch carried on the wrist [3].

Hawk-eye System is based in computer vision as well Hawk-eye[1] uses between six to ten cameras spread through the stadium. Usually used in tennis, badminton, football and cricket the system depends on the cameras' triangulation of the ball at all times through the pixel coordinates of the ball in each frame. It then uses this information to create a 3D view of the ball flight storing it a database. This system is very similar to the GoalControl system because it also allows for 3D representation of the ball's flight and even outputs several game statistics graphically [4] as shown in figure 2.2.

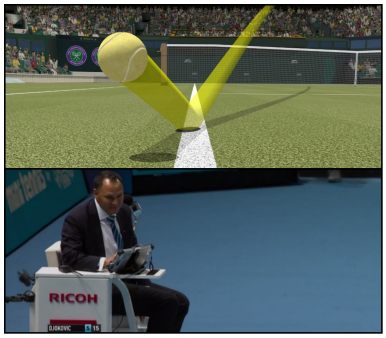


Figure 2.2: Relying on computer vision, the system allows for a 3D reconstruction of the game plays to check for example if the ball landed inside or outside the limits of the tennis court (top image). The referee receives information about the game plays through a tablet or a wristwatch depending on the sport (bottom image) [4].

[1] After a few other systems were trialled by UEFA, Hawk-eye was used for the football tournament Euro 2016.

2.1.2 Magnetic field based systems

Cairos-Adidas System is a system that relies in magnetic field sensors. The whole ball had to be redesigned because it contains a sensor that detects the magnetic field created by goal and must be protected from high force impacts. An example of the ball is presented in figure 2.3 [2]. Cables with electric current are buried behind the goal line to form a grid. The sensor in the ball measures the electrical field transmits the information to a computer that processes it and determines if the ball did pass the line of goal or not. Some critics to the system included criticism about the game delay that could be caused by the use of this system even though the manufacturer claims that the process is almost immediate.

Figure 2.3: The image presents the ball with the sensors inside that is used by the Cairos-Adidas.

GoalRef System is based in a magnetic field as well as Cairos-Adidas system however instead of having a sensor in the ball, the goal posts create a low frequency magnetic field that is monitored by coils in the posts and crossbar. Disturbances in this magnetic field are monitored by software. The system uses the same way of alerting the referees of the decision as the GoalControl and Hawk-eye systems, a wristwatch [5]. In figure 2.4 it is presented a schematic of the system with the stages of the ball detection.

2.1.3 Other systems

None of these systems presented in the previous subsection cover a basketball game, however some papers have been published containing research in terms of player position detection [6] or automatic detection of goals in basketball videos [7]. An example of the system at work is presented in figure 2.5. It can be seen that the detection of the limits of the field is performed before the players are detected, creating a region of interest for the next step (top right image). The next step is the detection of the players inside the field (bottom image). This system works on broadcasted images of the game, using only one camera.

[2]Image adapted from https://en.wikipedia.org/wiki/Goal-line_technology

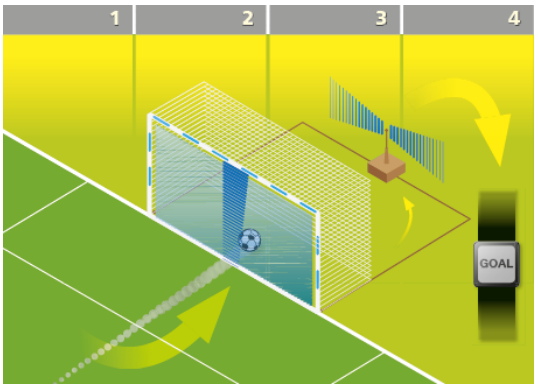


Figure 2.4: The steps necessary for the referee to receive information about the goal. The ball approaches the goal posts *(1)* disrupting the magnetic field *(2)*. The monitoring system detects the disturbance *(3)* and sends the information to the referee's wristwatch *(4)* [5].

Figure 2.5: This framework automatically detects the player position in the game of basketball starting by finding the limits of the field (top right image) and only then the players (bottom image). The system is used in game broadcasts with only one camera [7].

There have also been a few patents filed for systems that detect goals in several types of sports such as the one present in [8] that even though it expired in 2010 consists of a system with two cameras placed in the backboard over the basketball hoop and under it. The concept of the system is shown in figure 2.6 where the position of the cameras is shown placed on the basketball backboard. There are two cameras, one placed over the basketball hoop and another under it. The ball is detected when intersecting the detection planes of each of the cameras.

2.2 Previous Work in Robotic Soccer

CAMBADA is the RoboCup middle-size league soccer team of the University of Aveiro that has been competing worldwide since 2003 [9].

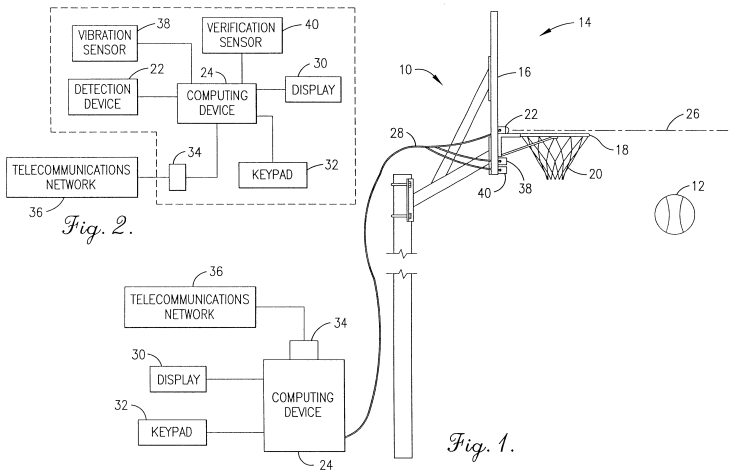


Figure 2.6: This is a basketball score-keeping aparatus for detecting and tracking shooting statistics of a player shooting a basketball at a basketball goal. It detects the first position where the ball crosses a horizontal detection plane as it ascends and a second position as it descends [8].

Figure 2.7: These are the robots that compose the team [9].

In 2014 the team moved to its own laboratory equipped with its own soccer field, shown in figure 2.8, where they could work on the several components of the robots, presented in figure 2.7, from hardware to software.

Each of the robots from the team extracts information about the world, such as the position of obstacles, other players and the ball, by using a camera. The current version of the vision system uses an omni-directional setup based on a catadioptric configuration implemented with a gigabit ethernet camera and a hyperbolic mirror [9] that can be seen in figure 2.9.

Figure 2.8: This is the field where the robots are tested and where this thesis was developed.

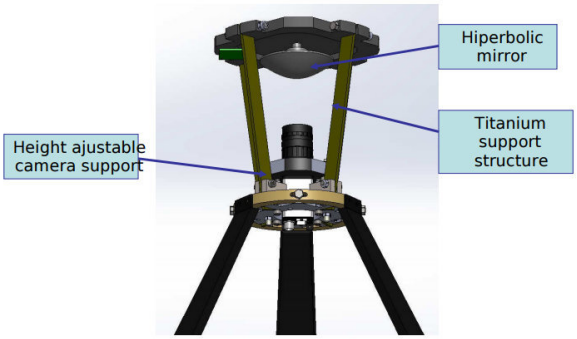


Figure 2.9: This is the setup of the camera component of each robot. The camera is capturing the image projected in the mirror so that the robot can see the entire field. The resulting image is then processed to remove areas that are not of interest, such as outside the limits of the field and the captured images of the titanium support. The image was adapted from [9].

2.2.1 UAVision Library

After the images are captured, they must be processed and with this purpose UAVision was developed. UAVision is an *OpenCV* based color detection library that integrates several different components that transform and analyze the captured frame pixel by pixel and allow to perform color segmentation [10]. The components from this library used in this thesis are explained in more detail in subsection 4.1.2.

2.2.2 Monitoring Robotic Soccer

Regarding ball position estimation and ball representation in 3D a master degree thesis was developed last year [11] in the IRIS Lab at the University of Aveiro. The thesis was based on the development of a system that allowed to monitor objects using several RGB cameras,

with a possible application in the monitorization of robotic soccer matches. Several methods to solve this challenge were presented and the method chosen was object triangulation using between two and four cameras.

Using two cameras for the triangulation, there was a study done to find the angle between cameras that had the best results. To study the effect of the angle between cameras two tests were performed. In the first the error in the position of the ball calculated by the system was studied for several angles between cameras. The angles used ranged between 0 and 180 degrees and several camera configurations were used to test each of the angles. In each of the configurations the ball was placed in several different positions on the field. In figure 2.10 the several camera configurations used are presented for the case where the ball is at the center of the field $(0, 0, 0)$. Measurements of the error for each one of the camera configurations for the ball detection are then presented in figure 2.11.

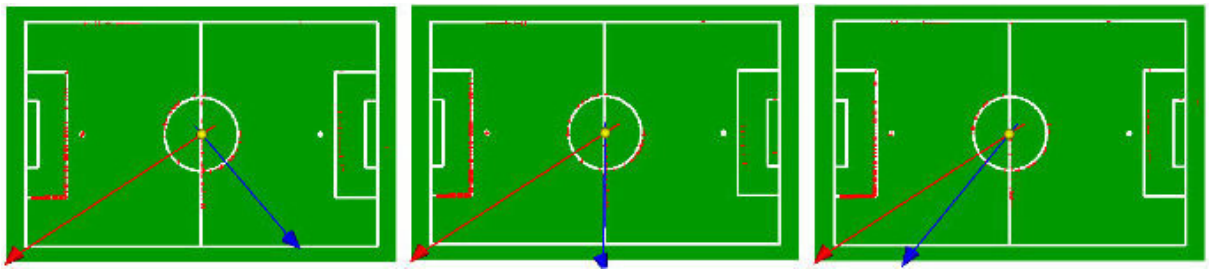


Figure 2.10: Camera configuration used to verify the impact of the angle between the two cameras [11].

In the second test, a detection error was forced into the system. The effects of the variation of the error were recorded for several different angles between the cameras. In both tests it was concluded that, to achieve an accurate and reliable system, triangulation using only two cameras was enough as long as the angle between the cameras was between 40 and 120 degrees.

Camera calibration was a major part of this work, and so, the effects of a good or bad calibration were studied as well. It was concluded that a good intrinsic calibration is more important than an extrinsic calibration for the system to be accurate. The results of a good and bad calibration are shown in figure 2.12. In a good calibration (right image of figure 2.12), the reprojection of the points should match the white lines of the field.

Processing time is key in robotic applications so the thesis also had a study about the processing time that was required to obtain the necessary information. The conclusions were that the system had a processing time acceptable for the applications it was designed for.

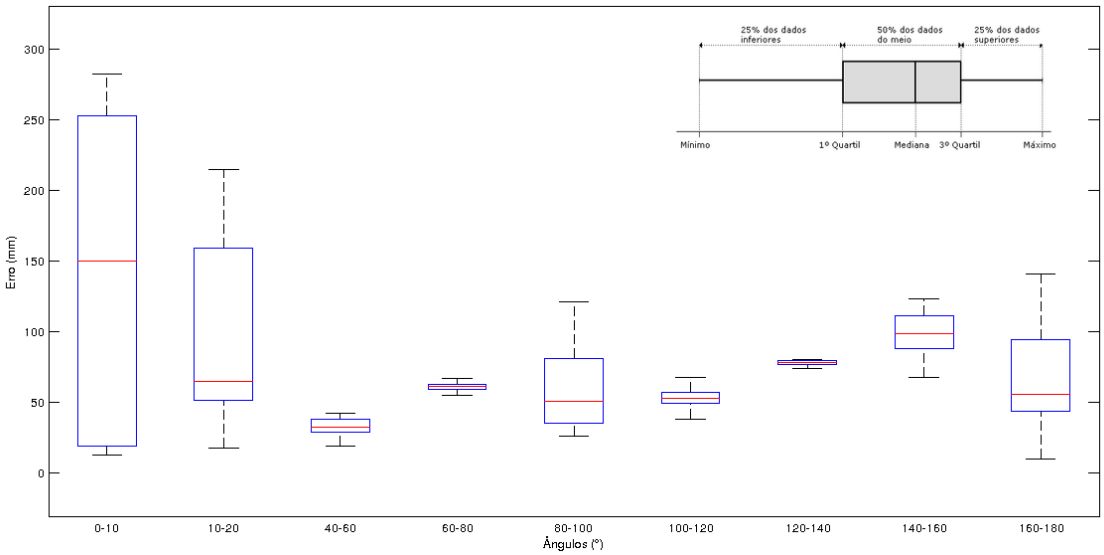


Figure 2.11: Error variation for the different camera angles, ranging between 0 and 180 degrees [11].

(a) (b)

Figure 2.12: For the same extrinsic parameters, in image *(a)* there is an example of a poorly performed calibration and in *(b)* a well performed calibration. From the reprojection of the lines, it is visible that a good intrinsic calibration is a must have for the system to be accurate in calculating the position of the ball [11].

Chapter 3

Stereo vision system

A stereo vision system can integrate two or more cameras and it is used to extract 3D information of objects in the world from 2D images. It can therefore be used to obtain information about the object's position in the world. In this chapter, an introduction to digital cameras and stereo vision systems is made. Also, the more relevant characteristics of the cameras used are presented as well as the software developed to interact with them and use them in a stereo vision system.

3.1 Digital Cameras

The first camera known, *camera obscura*, dates back to 1021 AD and originated the pinhole camera model. This model describes a simple box-shaped camera with an aperture with the size of a pin on one of its sides that captured the light of the surrounding environment as shown in figure 3.1. The light is projected onto the other side of the box creating an image portraying the environment.

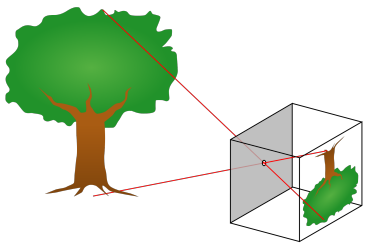


Figure 3.1: The light passes through the pinhole and the image is created on the other side of the box[2].

[2]Image adapted from `https://en.wikipedia.org/wiki/Pinhole_camera`

The captured image can be projected to a translucent screen to be viewed instantly (like it was in the *camera obscura*) or to a photographic film to store it.

The digital camera is a modern approach to the pinhole camera model where the photographic film is replaced by an electronic equipment that captures rays of the incoming light and converts them to electrical signals. This light detector can be of two types CCD or CMOS that can be represented as a 2D arrays of photodiodes in which each position stores information for each of the pixels.

The shutter is a component of the digital camera that allows to control the time that the light detector is exposed to the light rays (exposure time) that replaced a flap covering the pinhole on the older camera. When the shutter opens, light rays enter the camera and, after reaching the light detector, are converted to electrical signals. After the shutter closes, these signals are converted to digital values and stored. These values correspond only to the value of the brightness for each pixel and so it is required to add a filter to capture information about the colors. This filter *Bayer* matrix (shown in figure 3.2) that allows to store information about the colors red, green and blue by filtering the colors that are not of interest for each position.

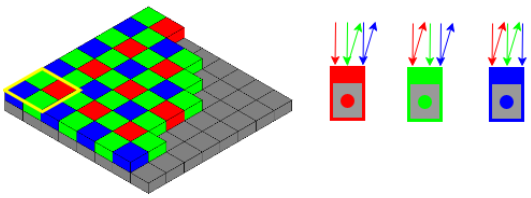


Figure 3.2: The *Bayer* filter is a mosaic of the three color channels red, green and blue. The green color is the most prominent color because the human eye is more sensitve to it. Each pixel of the resulting image is made of four of these array positions, one red, one blue and two green (marked by the yellow square).

3.2 Stereo Vision System

A stereo vision system is a system that integrates several digital cameras. For an accurate system, the cameras must be synchronized and aligned, either vertically or horizontally separated by known distance called baseline. Each of the cameras captures a different point of view of the world which, when processed with the other views can be used to extract 3-dimensional information about the world.

The most basic stereo vision system is composed of two pinhole cameras. As seen in figure

3.3 if only the image from the left camera is analyzed, there isn't enough information about the 3D coordinates of the object x because even if the object moves towards the camera there's no change in the pixel coordinates in the image.

However, if this information is combined with the information extracted from the right image, it is possible to triangulate the object in the world.
The points O and O' are the projection center of each of the images. When connected, the intersection of the line OO', also called baseline, with each of the image planes creates the points e and e'. e and e' are called epipoles even though in most cases these points are located outside the limits of the image. The lines l and l' are epipolar lines that aid in the rectification of the image.

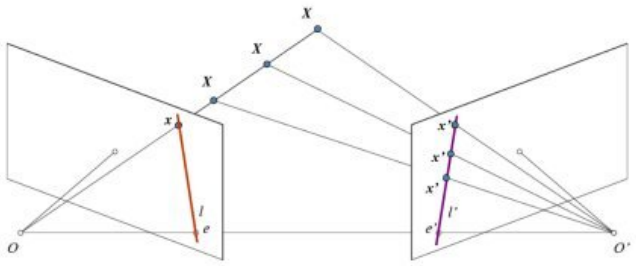


Figure 3.3: Epipolar geometry of a stereo vision system configuration[3]

In practice, it is required to find the epipolar lines and the epipoles which means that the cameras must be calibrated in order to find rotation and translation matrices of each camera and the intrinsics matrix as shown in section 3.2.1.

3.2.1 Camera calibration

Camera calibration is necessary to get the relation between the pixel coordinates in the digital image that correspond to the object and its 3D coordinates in the world. Without this step it is not possible to obtain the results sought. In order to obtain this correspondence, a set of parameters must be calculated, specifically the intrinsic and extrinsic parameters.

OpenCV[4] has specific functions to calibrate cameras. These functions are already optimized and tested making it easier to calculate the calibration matrices.

The intrinsic parameters are internal parameters specific to the camera that reflect characteristics such as lens distortion and lens focal length. In practice two different matrices are obtained, the distortion coefficients and the camera matrix.
For the distortion *OpenCV* takes into account the radial and tangential distortions. Radial

[3]Image adapted from `http://docs.opencv.org/3.1.0/da/de9/tutorial_py_epipolar_geometry.html`
[4]Library of programming functions aimed at computer vision originally developed by Intel

distortion results in what is commonly known by the fish-eye effect and corresponds to 3.1, 3.2, where r is 3.3. Tangential distortion is the distortion introduced by the angle formed by the lens and the imaging plane and is represented by 3.4 and 3.5. The formulas convert an undistorted pixel point *(x, y)* the pixel coordinates in the distorted image $(x_{distorted}, y_{distorted})$.

$$x_{distorted} = x(1 + k_1.r^2 + k_2.r^4 + k_3.r^6) \tag{3.1}$$

$$y_{distorted} = y(1 + k_1.r^2 + k_2.r^4 + k_3.r^6) \tag{3.2}$$

$$r = \sqrt{x^2 + y^2} \tag{3.3}$$

$$x_{distorted} = x + [2.p_1.x.y + p_2.(r^2 + 2.x^2)] \tag{3.4}$$

$$y_{distorted} = y + [p_1.(r^2 + 2.y^2) + 2.p_2.x.y] \tag{3.5}$$

So, *OpenCV* takes into consideration 5 distortion coefficients $(k_1, k_2, p_1, p_2, k_3)$. Besides these values a camera matrix, that holds information about the focal length f_x and f_y and the optical center for the camera c_x and c_x, is necessary.

$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

To determine the values presented above, correspondent to the intrinsic parameters of the cameras used, a small application was developed. The method used was the square chessboard method where several frames containing the chessboard in different angles and processed to find the inner edges as seen in 3.4. To achieve a good intrinsic calibration the chessboard must fill the most part of the frame.

Figure 3.5 presents the method used in the developed application to calculate the intrinsic parameters of the camera.

The extrinsic parameters are the camera position related parameters that translate to the camera rotation and translation in the world, using as reference known world points. Both rotation and translation of the camera are composed of 3 values (r_x, r_y, r_z) and (t_x, t_y, t_z) respectively.

For the extrinsic calibration of the cameras a small program was created where, based in some manual point correspondences, the values correspondant to the extrinsic calibration

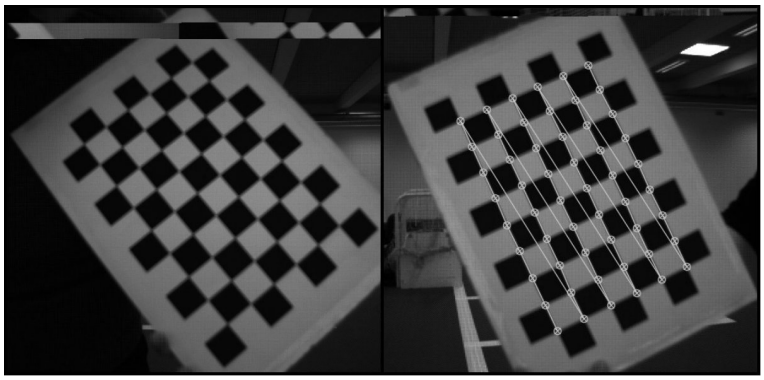


Figure 3.4: This is the chessboard used for the intrinsic calibration (left image). The chessboard is captured by the camera and its inner corners are found (right image).

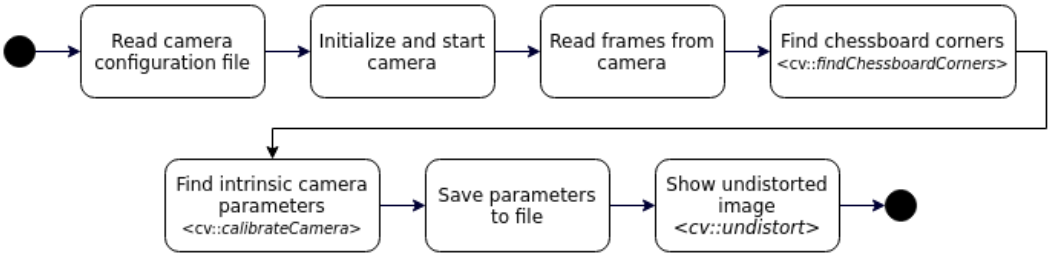


Figure 3.5: Diagram for the intrinsic calibration process that finds the chessboard corners using the function from *OpenCV cv::findChessboardCorners*, calculates the intrinsic parameters, stores the in a XML file. After these steps are complete the image of the camera is undistorted using *OpenCV*'s function *cv::undistort* and displays it to the user.

are calculated. A diagram containing the steps used to perform the extrinsic calibration is presented in figure 3.6. After the camera is connected, the user must click on several points on the image that correspond to known in world coordinates and must explicity specify the world point that it is referring to. For this, the world coordinates of the field corners were determined and stored. To make the user's job easier, each of the known world points is assigned a number as presented in appendix B and the user is asked to input the number of the corner. The image points that the user clicked are undistorted using *OpenCV*'s *cv::undistortPoint*. Using *OpenCV*'s function *cv::solvePnP* that, given the undistorted image points and the corresponding world points, calculates the rotation and translation vectors. These vectors are saved to the same parameters file that already contained the intrinsic parameters. An example of the usage of the program can be found in figure 3.7.

To validate the accuracy of the calibration the white pixels of the field's lines were detected, using the UAVision library (described in section 2.2.1), and projected onto a model of the

field built in a scale of 1 pixel for 20mm. After the projection of the points it is possible to calculate the calibration error by calculating the distance between the white lines in the field and the detected white lines.

To calculate the resulting error, a LUT was used where in each pixel of the build field model, the distance to closest white pixel is stored. The results of this phase are presented in chapter 5.

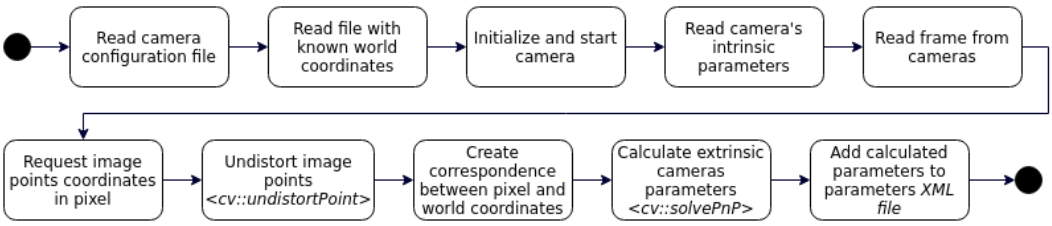


Figure 3.6: Diagram for the extrinsic calibration process that calculates the rotation and translation of the camera in the world.

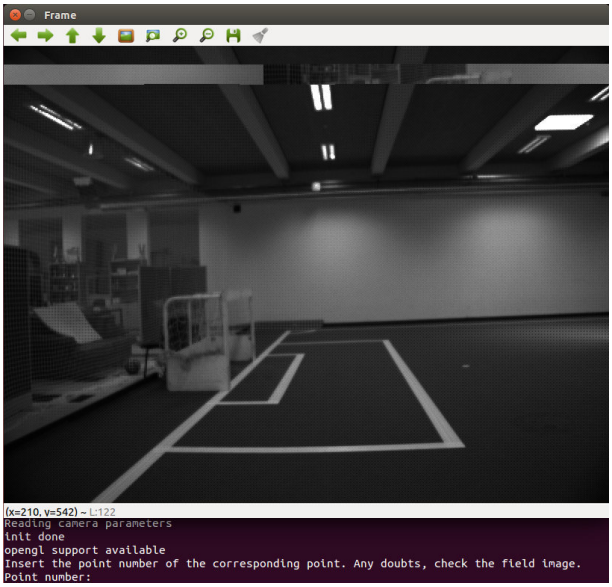


Figure 3.7: The user is provided with the frame captured from the camera. The points of the field should be picked on the image and the world point must be inserted to create the correspondance between the pixel coordinates and the world coordinates.

3.3 Object Triangulation using Digital Cameras

In order to have triangulation between the cameras in the work developed in this thesis, a system was built. The system is composed of two cameras, three computers, a router and

a *RaspberryPi* board.

Two of the computers serve as clients to the third computer, the server. Each one of the clients is responsible for acquiring a frame from the corresponding camera and process it in order to get the world coordinates for the ball using the program presented in 4.1.3. These coordinates are then sent to the server that is in charge of putting the information together and displaying it.

To handle the communication between clients and server TCP sockets were used and a specific message object was created that contained the coordinates calculated. This method was chosen over an already existing client-server solution because the complexity of the message objects wasn't worthy of implementing a more complex solution. Also, at this point, the information transferred between clients and server, only is transmitted in the local network, without the need of accessing the internet.

To handle the messages in the server, each of the entities perform the operations represented in figure 3.8.

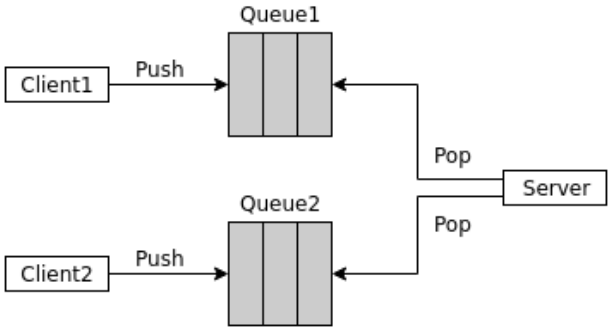


Figure 3.8: Both the clients and the server access the queues of messages used. Each one of the clients accesses one message queue in order to simplify the operations on the server side. When both the queues contain a message, the server retrieves both messages and processes them.

Each of the clients is only permitted to access its corresponding queue to simplify the operations on the server side. The client, as soon as it has the coordinates available, sends them to the server in the form of a message object and it is added to the corresponding queue. When both queues have messages, the server retrieves one message from each one of the queues and proccesses the information. Since both threads of client and server access the same queue, there was the need to create a thread safe queue that would prevent concurrency problems. This was done using a mutex for each of the queues that blocks the access to an entity if there is already another one accessing it.

The code for this queue is presented in appendix C. The size of each one of the queues is

limited to three messages. This was done so that the information processed by the server was the most recent. Should the synchronization module experience some kind of problem, since the system works in real time, the limited size of the queues helps prevent the system from trying to analyze either frames too far apart from each other (in a temporal matter) or frames that are old and contain outdated information.

The camera captures the frame of the scene and sends it directly to the computer to be processed. After the ball is detected in the frame, and the pixel coordinates of the ball are estimated, the world coordinates for the ball are calculated. Notice that the world coordinates calculated are in fact the coordinates of the projection of the ball on the working plane (the ground) as seen in figure 3.9. This happens because it is not possible to obtain 3D coordinates of the ball itself from 2D coordinates without additional information such as the ball's height (h).

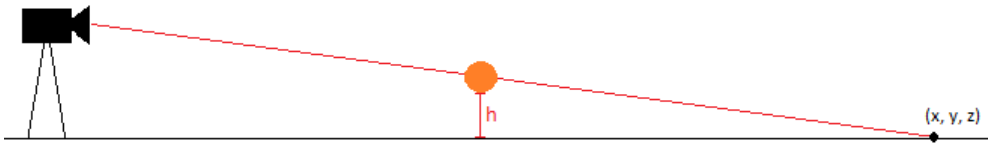


Figure 3.9: A client connected to a single camera does not calculate the actual world coordinates for the ball, but instead, the projection of the ball on the working plane. If the result from only one camera is analyzed, the higher the ball is, the further away the calculated coordinates will be from the actual ball.

This means that the coordinates (x, y, z) calculated by one camera do not correspond in fact to the ball coordinates and so, the information from both cameras must be analyzed at the same time. So, the computers connected to each one of the cameras, send the calculated point to a third computer that serves as the server for the system and where the information is put together.

For the actual world coordinates of the ball, a vector that connects the optical center of each of the cameras to the corresponding detected world coordinates is drawn. The coordinates of the ball in the world are then calculated by calculating the intersection point of both vectors as shown in figure 3.10.

3.4 Point Grey Zebra 2 Digital Camera

Besides the commercial systems presented in this chapter, there are others not refered here because they are very similar to these either using computing vision or magnetic field sensors. To track the ball in the field two digital cameras were used, more specifically two PointGrey

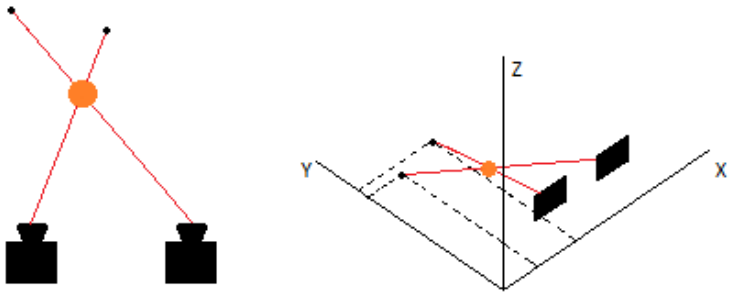


Figure 3.10: Ball coordinate calculation through the intersection of the vectors that connect each one of the cameras to the corresponding ball coordinates calculates.

Zebra - ZBR2-PGEHD-20S4C-CS [12] shown in figure 3.11. These cameras are used for many applications such as traffic control, stereo vision and others. There is a good compromise between resolution and framerate which is an important factor in fast paced game of basketball for example. They allow a *Gigabit Ethernet* connection to transfer frames between the camera and the computer. In comparisson to an USB connection, greater distances are supported without loosing information.
The cameras must be powered with a power source with a power ranging between 8 and 30V either through the 4-pin *Phoenix Connector* or the *GigE* interface.

Figure 3.11: This is the camera used. Zebra2 cameras are designed to fulfill the needs in traffic monitoring applications, surveillance, etc. This image was adapted from [12].

3.4.1 Camera Drivers

The manufacturer of the camera presented above provides a SDK to interact with and control the camera.
In this case, an interface from *UAVision* library was implemented so that several features such as *frame rate*, *gain* and *brightness* could be controlled.

This interface was implemented earlier in [11], however some changes had to be made in order to integrate trigger support. The order of the operations when initiliazing and connecting the camera had to be rearranged so that the camera would connect correctly to the application. The configurations of the camera had to be changed so that the framerate would increase and the processing time for each frame was smaller. Also, configurations for the trigger, such as specifying the trigger mode and the timeout, were added so that the camera would wait for the external trigger signal to fire. These changes are documented in appendix A.

The camera's framerate is greatly affected by the video mode of the camera, image and pixel formats. Hence the configurations applied to the cameras must be chosen carefully. Zebra2 cameras allow for several different video modes that determine how the camera captures the image. All modes allow the use of a region of interest (ROI) but only a few perform pixel subsampling. Pixel binning[5] is the aggregation of pixels and determines how the camera looks at the pixels in the image. This means that the camera will add, average or ignore neighbour pixels increasing the framerate at the cost of loosing resolution. There are two types of binning vertical and horizontal that indicate how the pixels are aggregated and there can be groups of 2 or 4 pixels. There is also an operation subsampling that ignores some pixels. A representation of the operations is presented in figure 3.12.

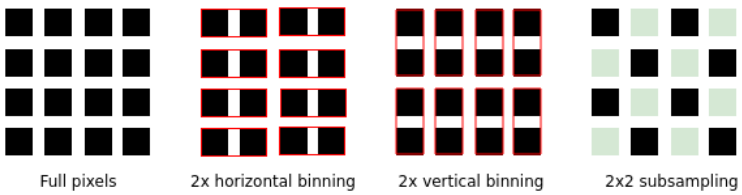


Figure 3.12: Pixel operations available in Zebra2 cameras. Their use depend on the video mode chosen because not all modes allow all operations [13].

The presented cameras have several video modes depending on the purpose of the application and the region of interest for the user that are described in table that follows [6].

[5]The term is used in the Zebra2 camera's technical reference and it will be used to refer to pixel aggregation
[6]The table was adapted from [13]

Mode	Description	Observations
0	All pixel scan (allows for ROI)	When ROI is applied, allows higher framerates.
1	2x/2x additive binning	Reduces the resolution in half in width and height. No framerate increase.
4	2x/2x subsampling	Poorer image quality than Mode 1, but higher frame rate. Available in RAW pixel formats.
5	4x/4x additive binning	Resolution reduced by a quarter in width and height. Not available in RAW pixel format.
6	4x/4x binning	Only available in monochrome pixel formats. Reduces resolution to one quarter in width and height.
7	Allows ROI but no binning (slower pixel clock, extended shutter)	Recommended for longer extended shutter times. There may be no framerate increase with the ROI size reduced.
8	Maximum resultion 1600x1200	Identical to Mode 0, but the maximum resolution is 1600x1200.

Table 3.1: Video modes available in the cameras used in this thesis.

For this application there was the need to choose a video mode that was supported by the cameras and that allowed a sufficient framerate. Keeping the framerate high is an important aspect to consider in systems that are to be used in realtime because of the processing time for each frame. So, out of the 7 modes available, mode 4 was chosen. This mode performs a 2 by 2 subsampling of the frame captured, decreasing the processing time and increasing the framerate. As such, the maximum image resolution is 812 by 612 that corresponds to half the maximum resolution the camera allows, 1624 by 1224. The other video modes are presented in more detail in [13].
Besides the video mode, pixel format also influences the framerate at which the camera processes the images captured. Pixel format dictates the encoding of the images, that can be in color or monochromatic, produced from raw image data. Some of the pixel formats available in the cameras used are RGB, YUV, Mono and RAW. Each of these pixel formats allow the number of bits per pixel to be changed between 8, 12 or 16 bits per pixel. RGB and YUV pixel formats require demosaicing on the processor side, whereas, RAW and Mono don't. RAW is a pixel format where the image data is Bayer RAW and is untouched by any on board processing allowing faster framerates. This pixel format allows for all the number of bits per pixel available. Since the processing time must be as low as possible, the RAW pixel format with eight bits per pixel was the choice for this application.

The IIDC IEEE-1394 Digital Camera Specification allows for the implementation of the Format7 custom image video modes. This means that the Format7 modes are defined by the manufacturer of the cameras which happens in the case of these cameras. Format7 modes, in contrast other configuration modes, allows for a precise control over the entire imaginng

pipeline such as image size, bining, color configuration and bandwidth consumption. In this thesis, to configure the cameras, the Format7 was used to set the video mode, image size and pixel format, which is shown in the code excerpt 1.

```
    bool CameraZebra::configureCamera(int width, int height, int vidMode, int
    grabTimeOut, bool trigger)
    {
        using namespace FlyCapture2;
        Error error;

        FlyCapture2::Format7ImageSettings fmt7ImageSettings;
        FlyCapture2::Format7PacketInfo f7pi;

        FlyCapture2::FC2Config config;
        FlyCapture2::Property prop;
        FlyCapture2::TriggerModeInfo triggerModeInfo;
        FlyCapture2::TriggerMode triggerMode;
        bool valid = false;

        fmt7ImageSettings.mode = MODE_4;
        fmt7ImageSettings.offsetX = 0;
        fmt7ImageSettings.offsetY = 0;
        fmt7ImageSettings.width = width;
        fmt7ImageSettings.height = height;
        fmt7ImageSettings.pixelFormat = FlyCapture2::PIXEL_FORMAT_RAW8;

    cam->ValidateFormat7Settings(&fmt7ImageSettings, &valid, &f7pi);
        if (!valid) {
            std::cout << "Unsupported image format settings" << std::endl;
            return false;
        }

        cam->SetFormat7Configuration(&fmt7ImageSettings, f7pi.
    recommendedBytesPerPacket);

      // ...
      return valid;
}
```

Listing 3.1: Necessary configurations of the camera to obtain the desired video mode.

While working with the cameras with the configuration presented above, a hardware limitation was found. On the top part of the frames captured, there are several rows of pixels that are a replica of rows from the lower part of the image. This was not only noticeable

in the tools developed, as shown in the right image of figure 3.13, but also in the capturing software provided by the manufacturer, as shown in the left image present in the same figure.

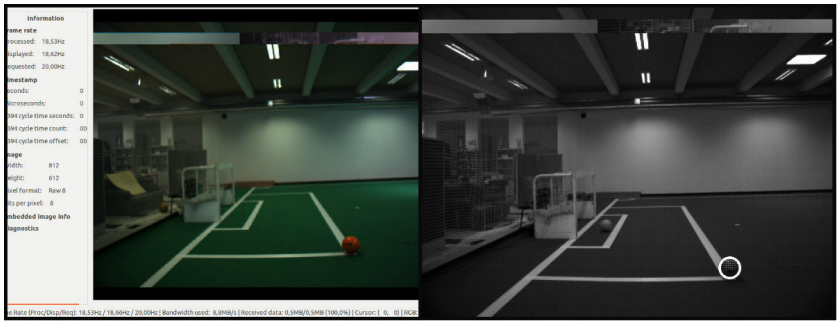


Figure 3.13: The rows that are replicated are noticeable here in the top of both images. The image on the left was captured using the software provided by the manufacturer with only the video mode and pixel format configurations set as intended. Whereas the right image is the result of the applications developed in this thesis.

Since the appropriate solution to this problem is out of the scope of this thesis, the solution implemented was to simulate a region of interest, while scanning the frame, that didn't include the affected area.

3.4.2 Synchronization

The camera used supports asynchronous triggering using an external trigger signal that is received through a 6-pin *Phoenix Connector*. A description of each one of the pins of the connector is presented in the table below.

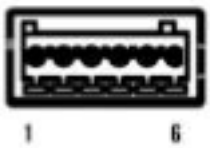
Schematic	Pin	Function	Description
	1	Opto In (IO0)	Opto-isolated input
	2	Opto GND	Ground for opto-isolated pin
	3	Opto Out (IO2)	Opto-isolated output
	4	GND	Ground
	5	RS485B	RS-485 signal (inverting)
	6	RS485A	RS-485 signal (non-inverting)

Table 3.2: Schematic for the pins of the 6-pin *Phoenix connector*. Each of the pin is described and its function indicated. The table was adapted from [13].

Pins 1 and 2 were used as the input for the trigger signal and the GND, respectively, as shown in figure 3.14. These pins were chosen because they are opto-isolated inputs which prevent damage to camera in case the input voltage is too high.

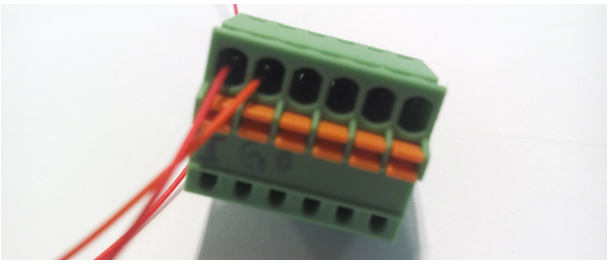


Figure 3.14: Visual record of the pins used to implement the synchronization module. Both pins are opto-isolated to prevent damage to the cameras caused by inputing higher voltage than specified in the technical reference [13].

To generate the trigger signal *per se* a *RaspberryPi* model B was used. This model comes with 512MB of RAM, one 100MB Ethernet port and two USB ports which, for this application, sufficed. The operating system used was *Raspbian* that is a free operating system based on *Debian* and it is optimized for the *RaspberryPi* hardware containing already a pre-compiled software bundle and over 35,000 packages [14].

The *RaspberryPi* outputs a PWM signal that indicates to the camera when to capture a new frame through its GPIO pins.
It outputs a maximum voltage of 3.3 volts and any output pin can be used. The physical pin 9 was chosen to serve as GND and the pin 11 as the signal output as shown in figure 3.15.

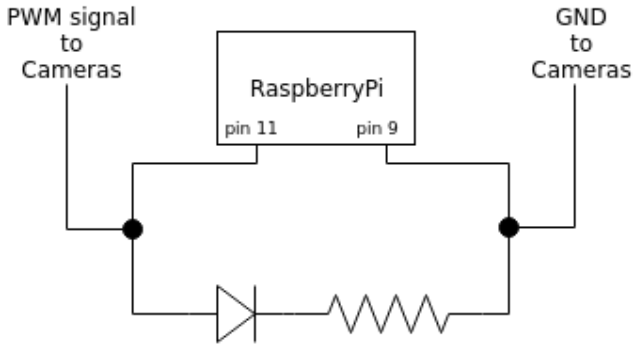


Figure 3.15: *RaspberryPi* component represented by a rectangle with indication to the pins that are being used. The camera's connection is represented by the two black circles and the LED represented there is merely used for debugging.

Software was necessary to generate the signal with the desired frequency and duty cycle and output it to the designated pins. As this task does not require any special operations and being that Python already provides the third party libraries necessary to directly interact with the *RaspberryPi* pins, Python was chosen over C++. The code for this script is shown

in the code excerpt 1.

```
import RPi.GPIO as GPIO
import time
import sys
import signal

def signal_handler(signal, frame):
    sys.stdout.write("Caught CTRL C: Clearning up...\n")
    my_pwm.stop()
    GPIO.cleanup()
    sys.exit(0)

if __name__ == "__main__":
    signal.signal(signal.SIGINT, signal_handler)
    GPIO.setmode(GPIO.BOARD)
    GPIO.setup(11, GPIO.OUT)
    my_pwm = GPIO.PWM(11, 76)
    my_pwm.start(95)
    while True:
        sys.stdout.write('.')
    signal.pause()
```

Listing 3.2: Implementation of the trigger signal generator in Python to be run in the *RaspberryPi*.

In order to define the frequency of the signal, the value should be specified in the initializer after the pin that will output the signal is indicated as well. This is done with *GPIO.PWM(pin, frequency)* replacing the values with pin number 11 and 76Hz, respectively. The duty cycle however is only specified when the start operation is called upon the PWM object, with *my_pwm.start(duty cycle)*, in this case, with the value of 95%.

At last, the camera must be configured to receive the trigger signal and capture the frame. There are several modes of trigger available in Zebra2 cameras that are explained next.

Mode 0 is the *Standard External Trigger* in which the camera starts the integration of the light from the external trigger input falling/rising edge. The integration time is defined by the exposure time.

Mode 1 is the bulb shutter where the integration of the light starts with the falling of the trigger edge and its duration is determined by the duration of the trigger's low state time. This mode required a long trigger's state that would influence the shutter time and this is automatically set by the camera.

Mode 13 is a reduced smear imaging mode that works in a similar way as mode 0 but the trigger input first activates a fast dump off the CCD followed by the frame capture which duration depends of the shutter settings.

Mode 14 is an overlapped exposure readout trigger that is similar to mode 0 but allows triggering at higher frame rates decreasing exposure precision.

And at last, mode 15 that is a trigger mode that captures sequences of frames. The number of frames is defined the user.

For this application there was the need for a trigger that allowed a fast capture of single frames for an unknown amount of time. Mode 14 is most suitable because it implements a standard external trigger allowing at the same time a higher framerate. A temporal diagram of how this mode operates is shown in figure 3.16. The mode uses the falling edge of the trigger signal to start the capture of the frame. After the frame is captured with a delay correspondant to the shutter time, the image is transfered over the network to the processing unit. If the image transfer was not completed, the mode allows for the next frame to be captured.

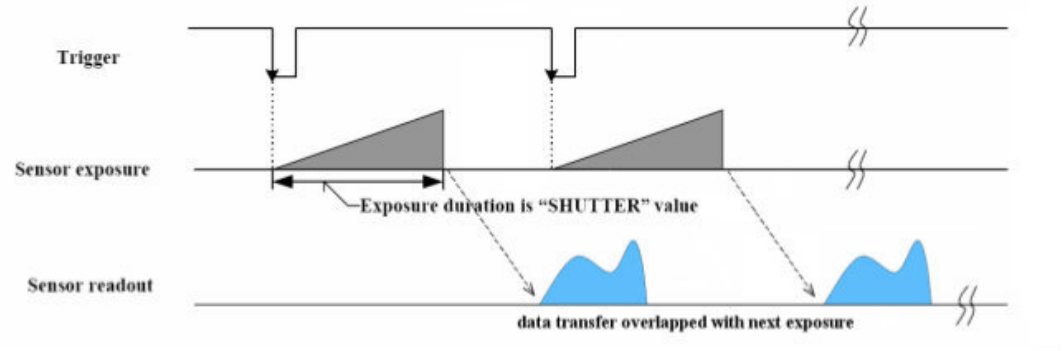


Figure 3.16: After the trigger signal is fired, the image is captured and sent over the connection. This mode allows for images to be captured while the previous image is being sent. This mode uses the falling edge of the trigger signal to start the capture [13].

Chapter 4

Ball position estimation

In this thesis the first step was to implement a stereo system in order to extract information about the ball's position. The process of distinguishing the ball from the rest of the world was automatized by color segmentation. Color segmentation segments the image by the pixel colors and it is useful to simplify the representation of the image making it easier to analyze. The ball is detect by analyzing the image and searching for its color.
In this chapter not only will these concepts be covered but also the creation of the 3D representation scene used to display the information to the user and the camera calibration that retrieves information about the cameras themselves such as distortions and misalignments.

4.1 Ball position detection

The whole system depends almost solely in the cameras being able to detect where the ball is in any frame. As referred in section 3.2.1, camera calibration is the first important step however in order to recognize the ball in the environment the method chosen was color segmentation.

4.1.1 Color Segmentation

Color segmentation was used in the process of distinguishing the ball in each frame by its color. In this study case the ball is orange, however it can be any color other than the color of the field, green. This is done by inputing the range of values for each color present in the environment. For this, an application was developed based on an already existing program from *UAVision*[1] that enabled the user to input the color ranges in HSV space color,

[1] *UAVision* is a library for color-coded object detection, that is currently being used by the robots of the team CAMBADA, participating in the Middle Size League of RobCup. The library has a modular design and it can be stripped down into several independent modules [10].

as shown in figure 4.1. This color space allows an abstraction over ligthing changes, creating robustness against the sun movement over the day for example. The application developed should run before anything else since it creates a configuration file for the camera that contains information not only about the color ranges but also the resolution preferred for the camera's frames and the pixel format.

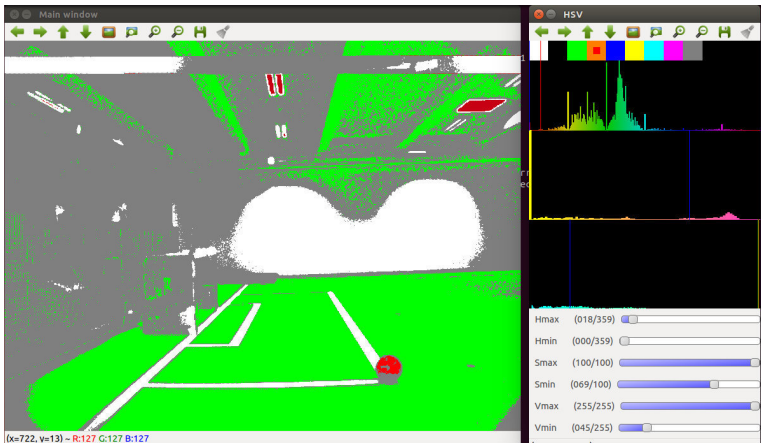


Figure 4.1: The user gets the frame from the camera and, for each color of interest, the range of the values of H, S and V components are chosen. After the values are chosen, they are stored in a configuration file that will then be read before detecting the ball.

4.1.2 Ball Detection

The UAVision library can be divided in several modules [10]. The color coded object segmentation module depends on three sub-modules. These modules were used in the ball detection operation of this thesis.

LUT is data structure that defines the color classes for a faster color classification. The processing time of the color classification is reduced because a runtime computation is replaced by an indexing operation. It is possible to use this component with several pixel formats, RGB, YUV[2] or Bayer adapting the LUT for each one of the possibilities. The table has 16,777,216 entries with one byte each. Each bit in the table tells if one other colors of interest is within the corresponding class or not.

ScanLines are lists of pixel positions in the image whose shape depends on the type of the ScanLine. There are three types available: radial, circular and linear (that can be divided into vertical and horizontal). Each ScanLine is used in the RLE module that

[2]Color space that translates Y into brightness and color UV into components.

produces a list of occurences of a specific color. The number of ScanLines defines the number of RLE lists produced. In the figure 4.2 there are examples of two types of ScanLines, horizontal and vertical.

Figure 4.2: This is an example of the ScanLines used. On the top image the horizontal type ScanLines are visible and on the bottom image, the vertical ones [15].

RLE is a component that for each *ScanLine* tries to obtain information about a specific color. It iterates through the ScanLine pixels to calculate the number of runs of that color and the position where they occur. The user can specify the color of interest that will be searched for and its neighboring colors. The final result is therefore a list of positions for each ScanLine where the color occurs as well as the number of pixels of each occurence.

Blobs are image regions that match the color specified in the *RLE*s. After being found, the blobs can be filtered by several characteristics such as area, size, width, etc. The first blob found is the first occurence of a specific color in the RLE and its center is the center of the ocurrence. A practical example of blobs usage to find the ball is shown in figure 4.3.

With the components reviewed, the logic pipeline for the ball's detection is shown in figure 4.4. It is shown that we start with the image captured from the camera that is transformed with the LUT into a color indexed image object. The processing is then performed on this image by the ScanLines and RLE to find the blobs candidate of being the ball.

4.1.3 Ball Position on Working Plane

Based on the ball detection algorithm, to retrieve the ball coordinates in the world, an application was developed that performed the entire image processing for the ball detection and the ball's position estimation. The configuration file is read to configure the camera

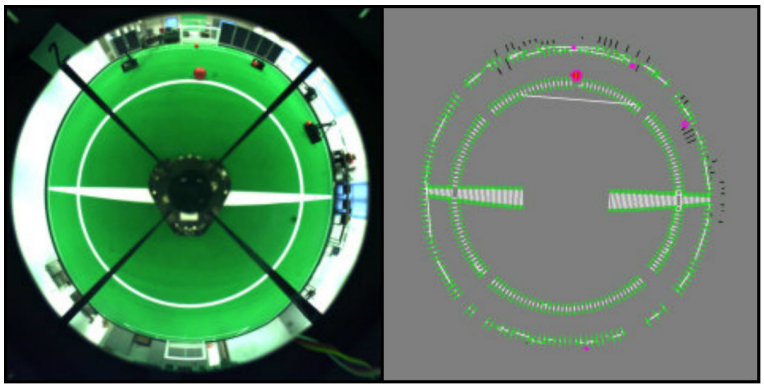


Figure 4.3: Practical example of the use of blobs on a CAMBADA robot [15].

Figure 4.4: This is the pipeline for the ball's detection. The red steps are the image processing steps *per se* whereas the others correspond to the image captured and the transformation necessary to improve processing time.

with the appropriate image resolution and pixel format so this file must be created with the application mentioned in 4.1. Reading the camera's parameters resultant from the camera calibration presented in 3.2.1 is a key step to calculate the ball coordinates and the *XML* files should also already exist. After these files are read and the content saved the image is processed accordingly to the diagram 4.5. The application displays the original image with a white circle around the detected ball to ensure the tracking is correct.

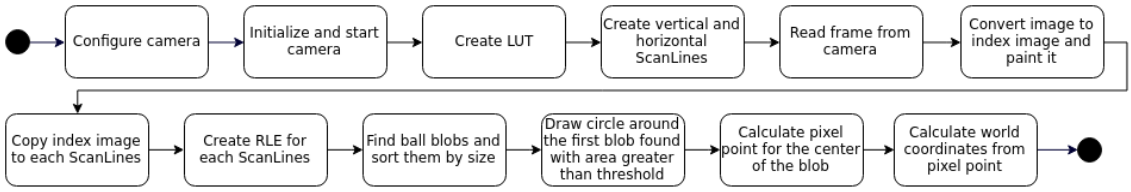


Figure 4.5: The logic behind the ball coordinates estimation is presented here with the different components from the library *UAVision*.

Note that, as explained in section 3.3, the world coordinates mentioned here are not the exact ball coordinates, but instead the coordinates of the projection of the ball on the working plane. In the diagram 4.5 there are some components that were developed in the UAVision libraries such as the LUT, *ScanLines* and RLE that are described in 2.

4.1.4 3D Ball Estimation

The system is made of 6 main components that connect with each other accordingly to the diagram present in 5.2.

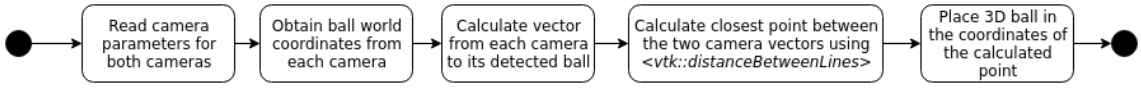


Figure 4.6: This diagram describes the process in the server that is performed to process the information received from the clients regarding the ball coordinates. The application creates a vector that connects each the point where the camera is to the point of coordinates received. After this, the closest point between the vectors is calculated and the ball is positioned in that point.

The diagram in 4.6 describes the process ran in the server. To put the information from both clients together the server takes both set of coordinates and calculates the vector that connects the camera to the point where the ball was detected. For this there was an application developed in *C++* and VTK that is an open source software system for 3D computer graphics and data visualization [16]. The server, after receiving the ball coordinates from the clients, creates a vector connecting the camera to its detected ball point. It then proceeds to calculate the intersection between the points. This is done because the coordinates received from the clients correspond to the projection of the ball in the ground plane and not the ball itself. Since cases where the vectors are paralel can occur, instead of calculating the point where the vectors intersect (that it would occur in the infinite), the closest point between the lines is calculated using VTK's function *vtk::closestPointBetweenLines*. This way if the vectors intersect the resulting point is the intersection point, otherwise the ball is placed between the lines.

Each one of the elements in the field has its own representive actor as shown in 4.7. The ball is represented as a sphere, whereas each one of the cameras is represented by a vertical vector. Each one of the cameras has its own color so that they are easy to distinguish.

4.2 Ball Trajectory Estimation

After having the position of the elements in 3D visualization, the next step was the trajectory calculation in order to find where the ball from a certain trajectory originated. The work performed here was adapted from the work on trajectory estimation present in [17].

Disregarding air resistance and other forces other than gravitational pull, the flight of a ball in three dimensions can be approximated by a ballistic trajectory in two dimensions. In

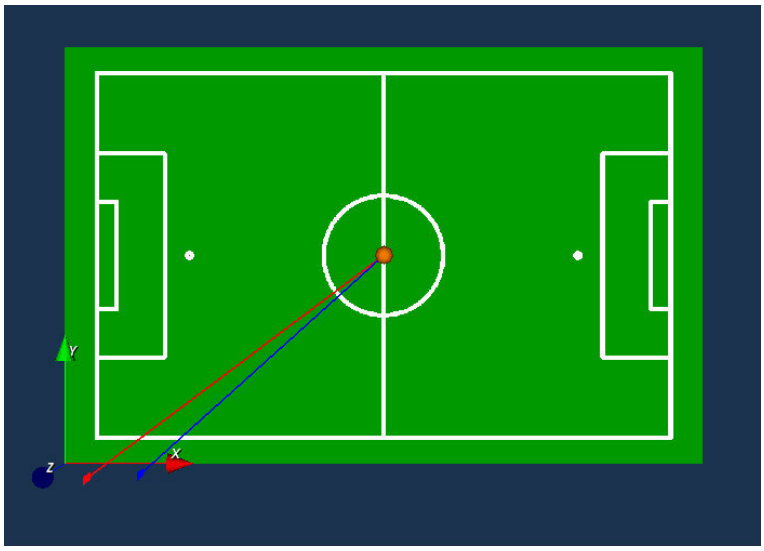


Figure 4.7: Each of the elements has its own actor. The cameras are here represented by vertical vectors, the ball by a sphere, the field by a plane with the field's texture and the camera's vectors by lines. Each of the cameras has its own color so that they can be distinguished (camera one is represented in blue and camera two in red).

these conditions, the approximation of the ballistic trajectory is a quadratic function.

The quadratic function is described by the equation present in 4.1.

$$y = c_0.x^2 + c_1.x + c_2 \tag{4.1}$$

In order to find c_0, c_1 and c_2, at least three points are needed since we have three incognitas. So supposing the three points are (x_1, y_1), (x_2, y_2) and (x_3, y_3) the equation for the quadratic function can be found by replacing the corresponding values and solving the system. In the implementation these points are the positions of the ball over time.

$$y_1 = c_0.x_1^2 + c_1.x_1 + c_2 \tag{4.2}$$

$$y_2 = c_0.x_2^2 + c_1.x_2 + c_2 \tag{4.3}$$

$$y_3 = c_0.x_3^2 + c_1.x_3 + c_2 \tag{4.4}$$

To simplify the steps performed this equation system can be replaced by operations with matrices. The matricial form present in 4.5 corresponds to the formula 4.2.

$$\begin{bmatrix} x_1^2 & x_1 & 1 \\ x_2^2 & x_2 & 1 \\ x_3^2 & x_3 & 1 \end{bmatrix} . \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} \tag{4.5}$$

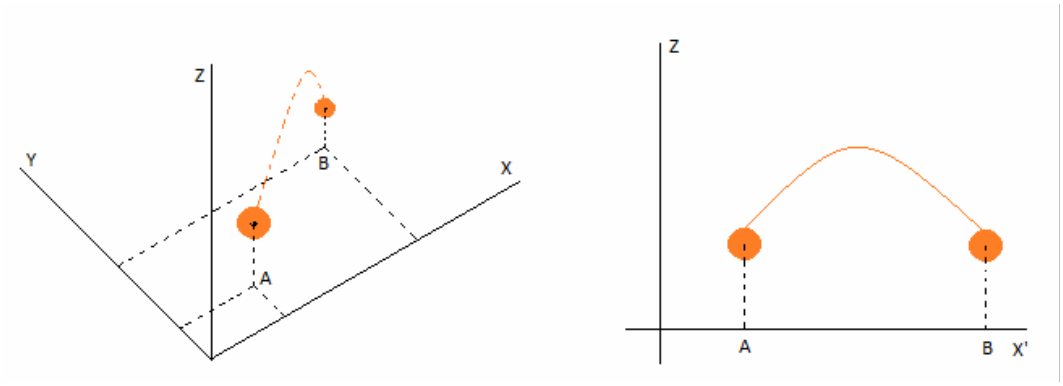


Figure 4.8: Even though the ball flight is done in three dimensions if the forces, besides gravitational pull, such as air resistance are disregarded, the trajectory can be approximated by a ballistic trajectory and thus described using a quadratic function.

After the values of the vector c are calculated, the equation of the quadratic function is complete.

The quadratic function has only two roots that in this case will correspond to the origin of the ball's flight and the destination, both on the ground plane, where $y = 0$. These can be found by calculating $y = 0$ through the function in 4.1. To calculate the roots of the function, the VTK *vtkPolynomialSolversUnivariate::SolveQuadratic()* function was used.

In practice, to implement the trajectory estimation, a history of the last positions of the ball must be kept and analyzed.
For every new position received in the server it is added to the history records. These records work along another auxiliary structure that stores information about the inclusion or not of the position in the trajectory. This structure has the same length as the history and must be updated alongside with it. There are three main actions performed to calculate the trajectory: *update_list*, *compute_trajectory_from_last_three* and *refine_trajectory* which code is present in appendix D. *update_list* is in charge of receiving the new position of the ball and updating the information related to the trajectory if it is a valid position. The new position is also compared to the last positions stored in order to decide if it belongs to the trajectory or not. This is done by calculating the length of the vector containing the origin of the current trajectory and the new position. A threshold for the allowed maximum distance must be defined in order to make this decision.
compute_trajectory_from_last_three is where the implementation of the trajectory estimation is done. The minimum number of points of a quadratic function needed to calculate the equation of the function is three points, so to do the math three points are taken into consideration at a time. The assumption is made that the antepenultimate point is the temporary origin of the

trajectory. The three points are then projected to the ground plane by giving them the value of 0 (zero) in the z axis. At this point, the distance between each of the other two points and the temporary origin is calculated making our referencial's origin at the penultimate point. As seen in 4.5 the vector x has to be calculated. In practice, using the *c++* template library *Eigen*[3] this vector can be calculated by using *MatrixBase::colPivHouseHol* and invoking *so.*. *refine_trajectory* is the function that is in charge of literally refining the trajectory in case there is already a valid trajectory. This function goes through all the positions in the history structure and recalculates the vector c with all the points that belong to the trajectory. If there is a point that does not belong to the trajectory, then its row in matrix X is set to 001.

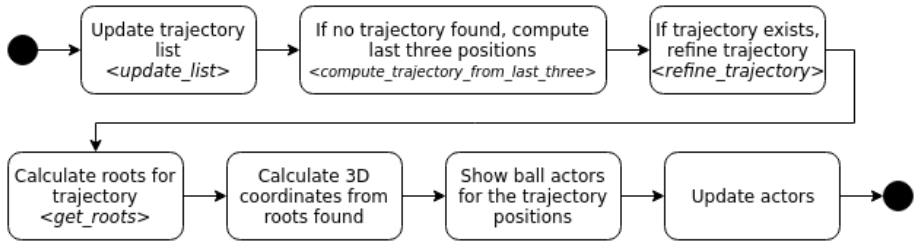


Figure 4.9: The steps in this diagram are simplified and serve to visually represent the logic used. The list of positions is updates as new positions arrive. If there is no valid trajectory, an attempt is made to compute a trajectory from the last three points. If the trajectory exists, it is updated (or refined) with the new position received. After, the roots for the trajectory are calculated as well as the 3D coordinates for each of the roots. The scene is then updates with the positions of the ball, calculated roots and the current position of the ball.

[3]Eigen is a C++ template library for liner algebra: matrices, vectors, numerical solvers and related algorithms [18].

Chapter 5

Results

This chapter presents the results of the developed stereo vision system and the tools described in the previous chapters. It contains several experiments and considerations about each set of examples, starting by the camera calibration, going through ball detection and ending with 3D ball trajectory estimation.

5.1 Experiment configuration

The stereo vision system created in this thesis is composed of two cameras aligned horizontally with a baseline of approximately 1.60 meters long as shown in figure 5.1.

Figure 5.1: The system developed for this thesis is composed of two cameras, three computers, a *RaspberryPi* and a router.

The 1.6 meters baseline was selected to evaluate the viability of the system to be used on the basketball backboard so that in the future the system could be integrated in it. The center of the system is the router that is represented in the diagram 5.2. All the components

are connected through *Ethernet* to it.

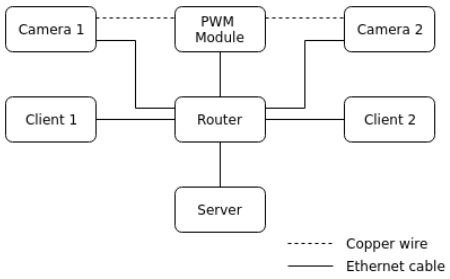


Figure 5.2: Complete system with two cameras connected to the PWM module as specified in subsection 3.4.2 and to two computers.

Even though the diagram present in figure 5.2 is the system configuration planned, the configuration used was different because of an hardware limitation in the router. The router used in this thesis had only four available *Ethernet* ports so the *RaspberryPi* was directly connected through an *Ethernet* cable to an outside computer in order to start the trigger signal. For the same limitation, one of the clients had to be run in the computer alongside the server.

5.2 Camera calibration

To measure the mean error for the camera calibration, a model of the field was created and was stored as an image. This model was previously created in [11] and was reused. Each pixel in the image corresponds to 20mm in the field. The tool developed, described in subsection 3.2.1, implemented a LUT that contained information about the distance of each one of the pixels to its closest white pixel. As a result three images were computed: an image that displayed the camera captured frame with the calculated points drawn on the detected white lines of the field, the information present in the LUT about the distances mentioned before and an image with the field model with the projected resulting points.
This process had to be performed on both cameras and it is expected that the projected points coincide with the white lines in all the images.

In figure 5.3 it is visible that the calibration was done successfully as the correspondences between the reprojected points and the field white lines are analyzed. The reprojected points further away from the camera have a slightly bigger deviation from the white lines than the others because the distance between the field lines and the camera is greater. On the image corresponding to camera number two (on the right) there are a few points that aren't on top of the lines near the goal posts limit. These points correspond to the reprojection of the

points to the ground plane where the goal was detected as a white area.

Figure 5.3: Results of the calibration of camera one (on the left) and camera two (on the right). The top sub image present on both side corresponds to the reprojection of the points on the white lines of the template field. Below, two images corresponding to the frame captured by the camera (on the left) and the result of the LUT with the reprojected points as well (on the right).

5.3 Ball detection

To test the detected position of the ball, the ball was placed static on the floor on specific points on the field that have known coordinates. To ensure that the system would work even when the distance between the ball and the cameras was increased, the points chosen covered most of the cameras field of view, varying the distance between the ball and the cameras.

Table 5.1 presents eight different tests with the ball placed across the field. The table contains information about the expected coordinates for the ball, the coordinates calculated by the system, the difference between each of the coordinates and the absolute error that corresponds to the euclidian distance between the points.

In figure 5.4 the steps of the ball detection are shown. The top left image is the result of the color classification performed witht the *UAVision* library mentioned in section 4.1. Whereas the top right image is the result of the ball detection by finding the blobs of orange in the frame captured by the cameras. The bottom image contains the 3D representation of the scene, showing the ball detected in red on the same position as the world ball one the field. In this experiment, number eight, the difference between the coordinates is 3.33 mm in the X-axis, 50.48 mm in the Y-axis and 20.55 mm in the Z-axis, which means that, the

	Expected (mm)			Resulting (mm)			Difference (mm)			Absolute
	X	Y	Z	X	Y	Z	X	Y	Z	error
1	-6812.5	-3187.5	0.0	-6823.67	-3225.28	121.68	11.17	37.78	121.68	127.9
2	-8312.5	-1687.5	0.0	-8310.54	-1709.64	132.84	1.96	22.14	132.84	134.7
3	-6812.5	3187.5	0.0	-6830.21	3221.07	100.77	17.71	33.57	100.77	107.7
4	-8312.5	5687.5	0.0	-8357.26	5221.27	224.86	44.76	466.23	224.86	519.6
5	-6812.5	-2567.5	630.0	-6809.82	-2600.91	628.45	2.68	33.41	1.55	33.6
6	-6812.5	3807.5	630.0	-6823.21	3807.15	660.34	10.71	0.35	30.34	32.2
7	-8312.5	1687.5	630.0	-8331.58	1568.12	675.99	19.08	119.38	45.99	129.3
8	-8312.5	-1687.5	630.0	-8309.17	-1737.98	650.55	3.33	50.48	20.55	54.6

Table 5.1: A number of experiments were performed by positioning the ball in known points on the field. This table presents the coordinates expected for the ball, the coordinates calculated by the system and the difference between these values.

system, with the ball at this distance from the cameras, is quite accurate considering that the unit used for the experiments is milimeters. Even though the experiments described in this section are meant to validate the estimation of the static ball position, the software used was the same as the one used for the trajectory estimation in section 5.4 and as such, some of the components shown in figure 5.4 are only explained in the next section.

Figure 5.5 presents the experiments where the ball is the closest (experiment number one) and the furthest away from the cameras (experiment number four), present in table 5.3, namely experiment one (on the left) and four (on the right). The difference between resulting coordinates and expected coordinates in experiment one are 11.17 mm for the X-axis, 37.78 mm for the Y-axis and 121.68 mm for the Z-axis. However, in experiment four, the difference is 44.76 mm for the X-axis, 466.23 mm for the Y-axis and 224.86 mm for the Z-axis.

In the table 5.3, the experiments performed where the ball was placed closer to the cameras are experiment one, two, five and eight. In general, the points closer to the cameras have a lower difference between the resulting coordinates and the expected. The point that is the furthest away from the cameras is used in experiment four and shown in figure 5.5. This is also the experiment with the highest difference between the coordinates expected and the resulting coordinates.

The difference between the coordinates is higher when the ball is placed further away from the cameras. This can be caused by the intersection point between the camera vectors moving away from the cameras as the ball also moves away or small errors propagated from the ball detection on each camera that will have greater influence in the result if the ball is

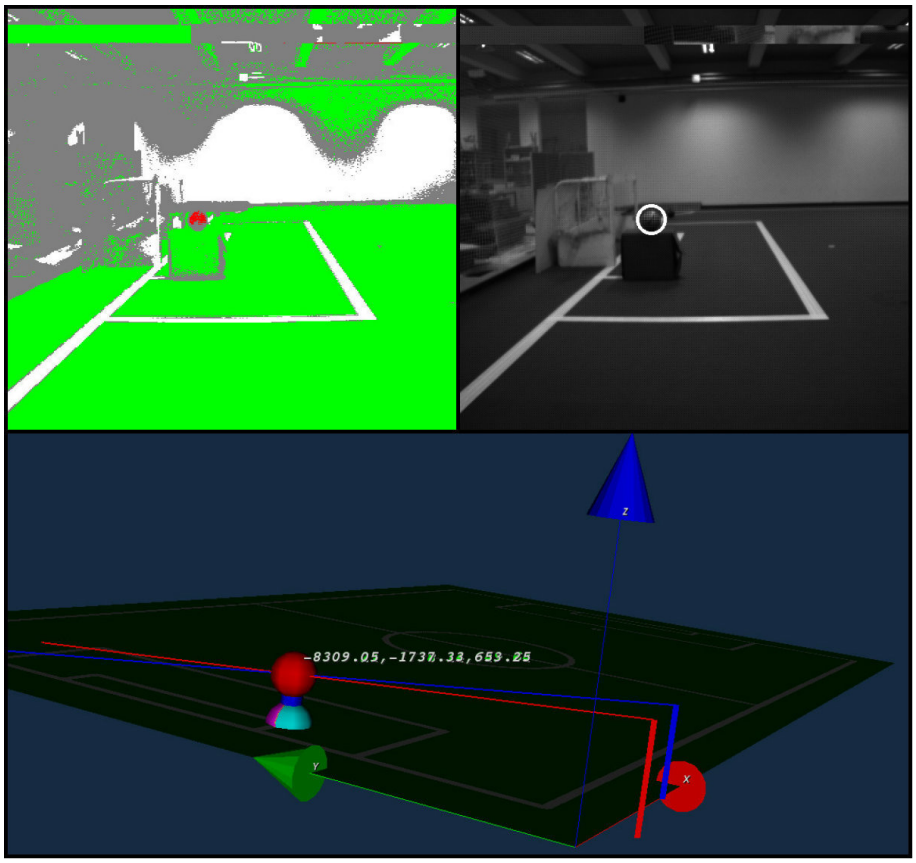


Figure 5.4: Images of experiment number eight. The top left image is the segmented image that represents the ranges of colors used for the color segmentation. The top right image is a frame captured from camera number one that shows the ball being detected on the field and marked with a white circle. The bottom image is the resulting 3D reconstruction of the ball on the field (the red ball), with the cameras represented as vertical vectors, and the axis X, Y and Z also represented by vectors and cones.

further away.

5.4 3D Ball trajectory

For the trajectory experiments, the whole system was put to the test. Images were captured from the synchronized cameras and then the ball was represented on the 3D scene along with the coordinates of the calculated point of origin for the trajectory.

In figure 5.6, there are four examples of the trajectory of a thrown ball in different directions. The white spheres represented the history of positions of the ball, whereas the red ones, the positions of the ball that fit the trajectory drawn in blue. Written in white are the

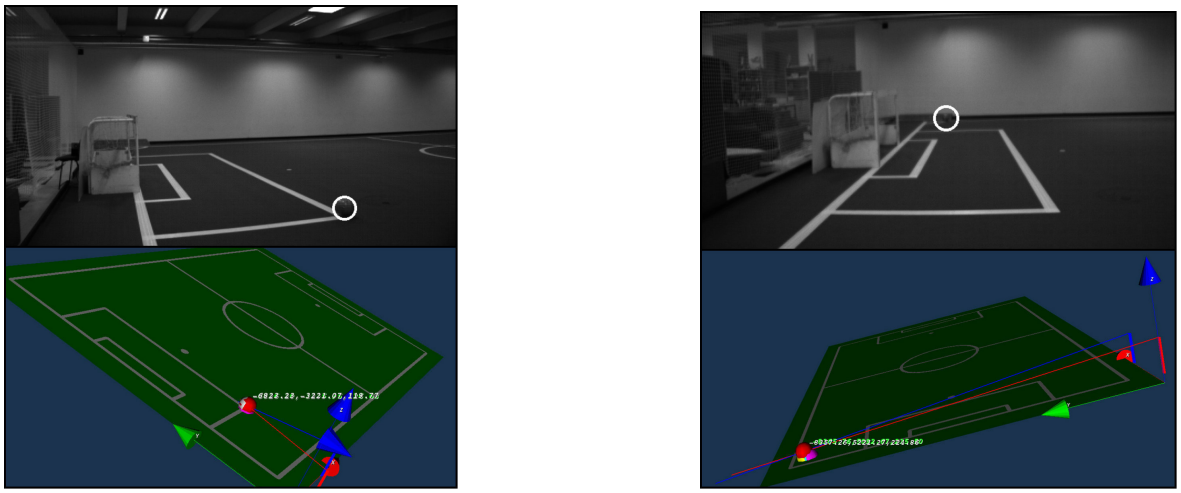


Figure 5.5: On the left is presented the result for experiment one, and on the left the result for experiment four.

coordinates of the calculated origin of the trajectory quadratic function, and in green, the lastest position of the ball captured on both cameras.

Even though there is no ground truth comparisson for the trajectory, it is based on the positions of the ball over time and the quadratic function mathematically calculated accordingly. As such, the difference between the expected coordinates for the ball and the resulting ones are the same as presented in section 5.3 for the static ball experiments.

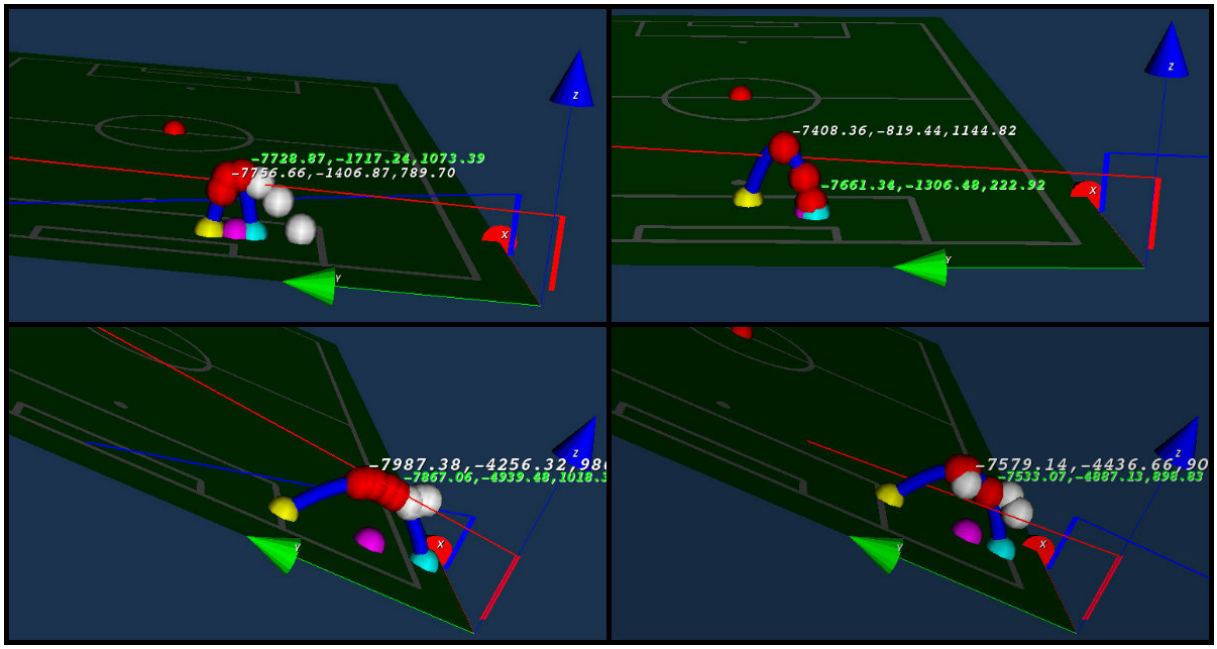


Figure 5.6: Examples of experiments performed on ball trajectories. The white balls are the positions of the ball over time, the red ones, the ball positions that fit the trajectory found. The blue ball is the estimated origin of the trajectory and the yellow, the estimated end of trajectory.

Chapter 6

Conclusions

Nowadays computer vision is an area that is emerging and every day new applications are appearing. Through sports there have been a number of developed systems that are being implemented in official games that help the referees make decisions about the score, mainly in soccer games.

This thesis attempted to study the viability of computer vision approach to basketball games focusing on an easily mountable system that was affordable and reliable. A stereo vision system composed of two cameras was chosen to achieve this. The small but existing complexity of the system arose from the processing time and power needed to complete this task. Attempting to divide the processing power needed by several affordable computers was a viable approach and so was the study of the image format that helped in terms of processing and transfer time as well as bandwidth.

New cameras were acquired and software was developed to use them, taking into consideration the image format referred in subsection 3.4.1. Moreover, a synchronization module was developed in order to use the external hardware trigger available in the cameras. Also, in order to have a viable stereo system, it was necessary to calibrate the cameras. After, taking into account the application in basketball, computer vision algorithms were developed to perform the ball detection and trajectory estimation.

Several applications were then developed to create a modular and easy to mount system.

1. The first application that was developed was one that allowed for the camera to be configured in terms of image format, trigger options and color ranges. The configuration file was created in this application and this should be the first application to be run.

2. For the camera intrinsic calibration there was an application developed that relied in the chessboard method to find the intrinsic parameters of the cameras. This application only

needs to be run once before the actual tracking and the 3D visualization. This creates a *XML* file that contains the matrices needed to represent the intrinsic parameters.

3. Still in the camera calibration module there is also an application that allows to discover the extrinsic parameters for the camera such as translation and rotation in relation to the world. This application should be run everytime the camera is moved from its place. The extrinsic calibration is done by having the user click in known points in the world and so there must always be a coordinates file that contains the ordered points for each world point. The parameter's file must already exist with the intrinsic calibration results or else the execution will fail.

4. After the previous steps the system is ready to start outputing the coordinates for the ball at any point in time. For this there was an application developed that if ran with the correct parameters serves as the client to the server. This application requires the configuration and parameters files to exist and contain the correct information.

5. The last application developed was the application for the 3D visualization of the ball and field which is the application required to run in the server, with the right input parameters. This takes care of the receiving the coordinates from both clients (each one of the cameras) and displaying the information in the 3D VTK scene.

As the system was complete, some experiments were made in order to validate the accuracy of the system. Since the trajectory estimation of the ball is based on the positions of the ball over time and mathematical calculations for the quadratic function that best fits these positions, the focus of the experiments was on the ball coordinates calculation. To test the calculation of the ball coordinates in the world by the system, the ball was placed in several points, on the field, with known coordinates. From these experiments, it is noticeable that the further away the ball is from the cameras, the higher is the difference between the expected values for (x, y, z) and the resulting for the same variables. This happens because the angle between the cameras should be higher and the baseline bigger to have camera vectors intersection more defined, even when the ball is further away.

6.1 Future Work

There are adjustments still to be made to what was developed here such as:

- The system as-is is complete however the idea of connecting it to a score board wasn't implemented as there were delays in the implementations and the validation of the ball entering the hoop was not implemented.

- The system does not contemplate the validity of the player's foot touching the lines in the field or not. This was not implemented as it wasn't the goal of this thesis because it would require another camera or another set of cameras mounted on the basketball dome.
- In terms of extrinsic camera calibration the user interaction is still required. The perfect solution would perform the calibration by itself by detecting the lines of the field and using a learning algorithm.
- The system is not fail safe and sometimes the connection to the cameras is not made instantaneously raising errors. The interface used to connect to the cameras should be reviewed and changed if possible to prevent these situations.
- The trigger is at this moment a separate module that has to be trigger outside of the main server application. The best way was to find a way to trigger it in the main application so that it was triggered when the clients were ready to fetch frames.

Bibliography

[1] R. Zabih, D. Scharstein, O. Veksler, V. Kolmogorov, A. Agarwala, M. Tappen, C. Rother, and R. Szeliski. A comparative study of energy minimization methods for markov random fields with smoothness-based priors. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 30:1068–1080, 2008. [Accessed 15-June-2016].

[2] Sports Partner International. About us. `http://www.sportspartnerinternational.com/?page=about-us`, 2016. [Accessed 15-June-2016].

[3] GoalControl GmbH. The goalcontrol products. `http://www.goalcontrol.de/en/products/`, 2016. [Accessed 27-May-2016].

[4] Hawk-Eye Innovations Ltd. Ball tracking. `http://www.hawkeyeinnovations.co.uk/products/ball-tracking`, 2015. [Accessed 27-May-2016].

[5] Fraunhofer-Gesellschaft. Operating mode – goal detection system. `http://www.iis.fraunhofer.de/en/ff/kom/proj/goalref.html`, 2016. [Accessed 27-May-2016].

[6] Z. Ivankovic, M. Rackovic, and M. Ivkovic. Automatic player position detection in basketball games. *Multimedia Tools Appl.*, 72:2741–2767, 2014.

[7] S. Nepal, U. Srinivasan, and G. Reynolds. Automatic detection of 'goal' segments in basketball videos. 72:261–269, 2001.

[8] R.R. Hampton. Basketball goal sensor for detecting shots attempted and made. `http://www.google.com/patents/US6389368`, 2002. US Patent 6,389,368.

[9] University of Aveiro IEETA. CAMBADA Robotic Soccer. `http://robotica.ua.pt/CAMBADA/index.htm`. [Accessed 16-June-2016].

[10] A. Trifan, A. Neves, B. Cunha, and J. Azevedo. Real-time color coded object detection using a modular computer vision library. *ACSIJ Advances in Computer Science: an International Journal*, 5, 2016. Issue 1.

[11] Fred Gomes. 3D objects monitoring based on multiple cameras. *Master thesis at University of Aveiro*, 2015.

[12] Inc Pont Grey Research. Zebra2 2.0 MP Color GigE / HD-SDI (Sony ICX274). `https://www.ptgrey.com/zebra2-2-mp-color-gige-vision-hd-sdi-sony-icx274-camera`, 2016. [Accessed 27-May-2016].

[13] Point Grey Research Inc. Point Grey Zebra2 PoE HD-SDI Digital Camera. *Technical Reference v2.1*, 2014.

[14] Debian. Welcome to raspbian. `https://www.raspbian.org/`. [Accessed 20-June-2016].

[15] A. Trifan. Time-constrained colored object detection for intelligent robots. *PhD thesis at University of Aveiro*, 2015.

[16] K. Martin, W. Schroeder, and B. Lorensen. The virtualization toolkit, 4th edition, 2006. ISBN 978-1-930934-19-1.

[17] P. Dias, J. Silva, R. Castro, and A. Neves. Detection of aerial balls using a kinect sensor. *RoboCup 2014: Robot World Cup XVIII*, pages 537–548, 2015.

[18] B. Jacob and G. Guennebaud. Eigen is a c++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. `http://eigen.tuxfamily.org/index.php?title=Main_Page`, 2016. [Accessed 29-May-2016].

Appendix A

Appendix A

A.1 Camera Header File

Here the methods implemented in the class are listed through the class's header file.

```
        /* Constructors */
        CameraZebra(FlyCapture2::IPAddress ipAddress, FlyCapture2::
MACAddress macAddress, FlyCapture2::IPAddress subnetMask, FlyCapture2::
IPAddress DG, int numCamera);
        CameraZebra();

        /* Destructor */
        virtual ~CameraZebra();

        /* Initializing the camera */
        int initCamera(CameraSettings *camSettings);
        int init(CameraSettings *camSettings);

        /* Starts camera */
        void startCamera(CameraSettings *camSettings);

        /* Stops the camera */
        void stopCamera(CameraSettings *camSettings);

        /* Stops the camera and finishes the connection */
        void shutDownCamera();

        /* Sets the camera parameters */
        void setParameters(CameraSettings *camSettings);

        /* Sets the value of one camera parameter */
        void setParameter(unsigned int parameter, unsigned value);
```

```
        /* Retrieves one or all the camera parameters */
        void getParameter(unsigned int parameter, double *value);
        void getParameters(CameraSettings *camSettings);

        /* Prints one or all the camera parameters */
        void printParameter(unsigned int parameter);
        void printParameters();

        /* Retrieves the ranges of one or all the camera parameters */
        void getParameterRanges(unsigned int parameter,ParameterRange
parameterRange);
        void getParametersRanges( ParameterRange parameterRange[] );

        /* Reads a frame from the camera */
        int readFrame(cv::Mat &image);

        /* Returns the number of cameras detected */
        int numCamerasDetected();

        /* Configures the camera in automatic or manual mode */
        void setAutoMode(CameraSettings *camSettings );
        void setManualMode(CameraSettings *camSettings );

        /* Forces the IP Address for a specific camera */
        void forceIpAddress();

        /* Configures the specified camera with the appropriate image size,
 video mode and trigger settings */
        bool configureCamera(int width, int height, int vidMode, int
grabTimeOut, bool trigger);
```

Listing A.1: Header file for the camera driver.

A.2 Camera Driver Implementation

The changes made to the class are presented below. Only the implementation of the methods that had modifications since [11] are refered.
This method initializes the camera, identifying it by the IP Address.

```
int CameraZebra::initCamera(CameraSettings *camSettings)
{
    using namespace FlyCapture2;
    using namespace std;
```

```
    using namespace uav;
    Error error;

    if (numCamerasDetected()==0){
        cout << "ERROR: No cameras connected" << endl;
        return -1;
    }

    if(camSettings->getCameraSetting(UAV_CAMERAUSED) == 0)  {
        startCamera(camSettings);
        camSettings->setCameraSetting(UAV_CAMERAUSED, 1);
    }
    else    {
        stopCamera(camSettings);
        startCamera(camSettings);
        camSettings->setCameraSetting(UAV_CAMERAUSED, 1);
    }

    camSettings->setCameraSetting(UAV_CAMERARUNNING,1);

    FlyCapture2::BusManager busMgr;
    guid  = new FlyCapture2::PGRGuid;
    error = busMgr.GetCameraFromIPAddress(iPaddress, guid);
    if (error != PGRERROR_OK)
        PrintError( error );

    error = cam->Connect(guid);
    if (error != PGRERROR_OK)
        PrintError( error );

    return 0;
}
```

Listing A.2: Changes made to the camera initializing method implementation.

In readFrame, the camera captures the frame, converts it to Mat image format and returns the image.

```
int CameraZebra::readFrame(cv::Mat &imageOpenCV)
{
    using namespace FlyCapture2;
    using namespace std;
    using namespace cv;

    Error error;
```

```
    if(!cam->IsConnected()) {
        cout << "Camera is not connected!" << endl;
        return -1;
    }

    Image rawImage;

    // Retrieve an image
    error = cam->RetrieveBuffer( &rawImage );
    if (error != PGRERROR_OK)
        PrintError( error );

    unsigned int rowBytes = (rawImage.GetBitsPerPixel() / 8) * rawImage.
GetCols();

    imageOpenCV = Mat(rawImage.GetRows(), rawImage.GetCols(), CV_8UC1,
rawImage.GetData(), rowBytes);
    return 0;
}
```

Listing A.3: Method implementation that grabs a frame from the camera and converts it to the OpenCV image format.

This method configures the camera with the given image size, video mode and trigger settings.

```
bool CameraZebra::configureCamera(int width, int height, int vidMode, int
grabTimeOut, bool trigger)
{
    using namespace FlyCapture2;
    using namespace std;
    using namespace uav;

    Error error;
    FlyCapture2::Format7ImageSettings fmt7ImageSettings;
    FlyCapture2::Format7PacketInfo f7pi;
    FlyCapture2::FC2Config config;
    FlyCapture2::Property prop;
    FlyCapture2::TriggerModeInfo triggerModeInfo;
    FlyCapture2::TriggerMode triggerMode;

    fmt7ImageSettings.mode = MODE_0;
    fmt7ImageSettings.offsetX = 0;
    fmt7ImageSettings.offsetY = 0;
```

```
        fmt7ImageSettings.width = width;
        fmt7ImageSettings.height = height;

        switch(vidMode)     {
            case UAV_GRAY:
                fmt7ImageSettings.pixelFormat = FlyCapture2::PIXEL_FORMAT_RAW8;
                break;

            default:
                cout << "Video mode not supported" << endl;
                return false;
                break;

        }

        bool valid = false;
        error = cam->ValidateFormat7Settings(&fmt7ImageSettings, &valid, &f7pi)
    ;

        if ( error != PGRERROR_OK)  {
            cout << "Failed to verify settings!" << endl;
        }

        if (!valid) {
            cout << "Unsupported image format settings" << endl;
            return false;
        }

        error = cam->SetFormat7Configuration(&fmt7ImageSettings,f7pi.
    recommendedBytesPerPacket);
        if (error != PGRERROR_OK)   {
            PrintError( error );
        }

        if(trigger) {
            if(grabTimeOut <= 0)
                grabTimeOut = 1000;

            prop.autoManualMode = true;
            prop.type = FRAME_RATE;
            cam->SetProperty(&prop);

            //Set Trigger
            error = cam->GetTriggerModeInfo(&triggerModeInfo);
```

```
        if(error != PGRERROR_OK)    {
            cout << "Failed to get trigger mode info!" << endl;
            return false;
        }

        if (triggerModeInfo.present != true)  {
            cout << "Camera does not support external trigger! Exiting..."
<< endl;
            return false;
        }

        //Get trigger mode
        error = cam->GetTriggerMode( &triggerMode );
        if (error != PGRERROR_OK)   {
            cout << "Failed to get trigger mode" << endl;
            return false;
        }

        triggerMode.onOff = true;
        triggerMode.mode = 0;
        triggerMode.parameter = 0;
        triggerMode.source = 0;

        error = cam->SetTriggerMode(&triggerMode);
        if(error != PGRERROR_OK) {
            cout << "Failed to set trigger mode" << endl;
            return false;
        }
    }

    if(grabTimeOut > 0) {
        // Get the camera configuration
        error = cam->GetConfiguration(&config);
        if (error != PGRERROR_OK)
        {
            cout << "Failed to get configuration" << endl;
            return false;
        }

        // Set the grab timeout to 2 seconds
        config.grabTimeout = grabTimeOut;

        // Set the camera configuration
        error = cam->SetConfiguration(&config);
```

```
            if (error != PGRERROR_OK)
            {
                cout << "Failed to set configuration" << endl;
                return false;
            }
        }

        return valid;

    }
```

Listing A.4: Camera configuration method implementation.

Appendix B

Appendix B

B.1 World coordinates for specific field points

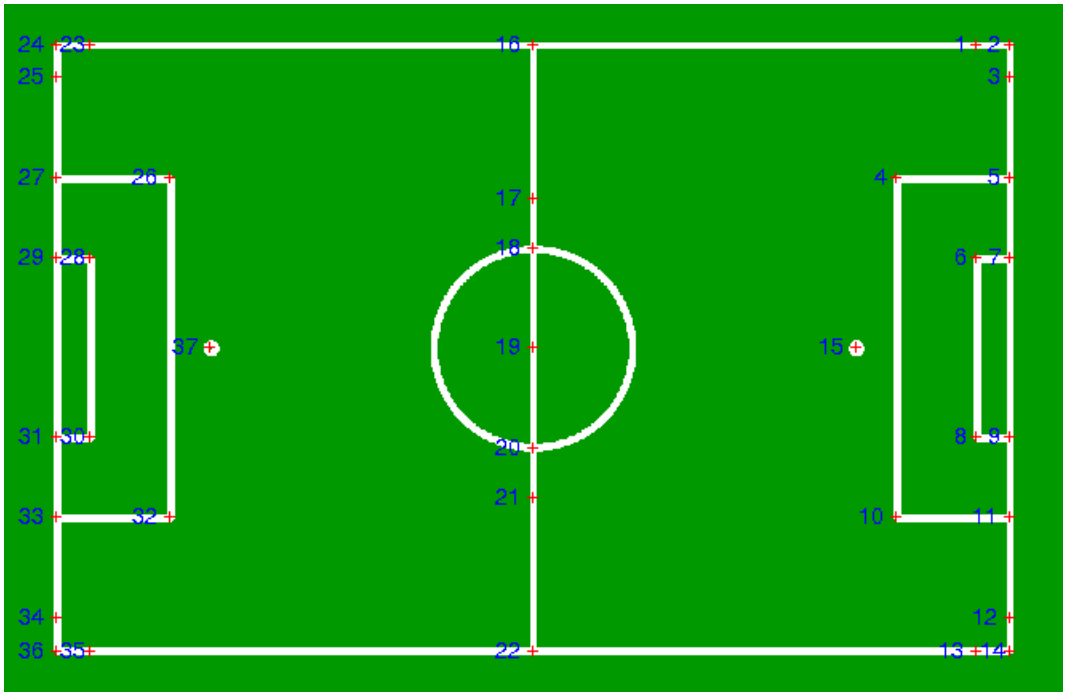


Figure B.1: Field points chosen for the extrinsic calibration [11]

Point nr.	X (mm)	Y (mm)	Point nr.	X (mm)	Y (mm)
1	8312.5	5687.5	20	0	-1875
2	8937.5	5687.5	21	0	-2800
3	8937.5	5062.5	22	0	-5687.5
4	6812.5	3187.5	23	-8312.5	5687.5
5	8937.5	3187.5	24	-8937.5	5687.5
6	8312.5	1687.5	25	-8937.5	5062.5
7	8937.5	1687.5	26	-6812.5	3187.5
8	8312.5	-1687.5	27	-8937.5	3187.5
9	8937.5	-1687.5	28	-8312.5	1687.5
10	6812.5	-3187.5	29	-8937.5	1687.5
11	8937.5	-3187.5	30	-8312.5	-1687.5
12	8937.5	-5062.5	31	-8937.5	-1687.5
13	8312.5	-5687.5	32	-6812.5	-3187.5
14	8937.5	-5687.5	33	-8937.5	-3187.5
15	6057.5	0	34	-8937.5	-5062.5
16	0	5687.5	35	-8312.5	-5687.5
17	0	2800	36	-8937.5	-5687.5
18	0	1875	37	-6057.5	0
19	0	0			

Table B.1: Correspondences between points numbers presented in B.1 and the coordinate values [11].

Appendix C

Appendix C

C.1 Concurrent message queue

```
#ifndef CONCURRENTQUEUE_H_
#define CONCURRENTQUEUE_H_

#include <queue>
#include <thread>
#include <mutex>
#include <condition_variable>

template <typename T> class concurrentQueue
{
 public:
  void pop(T& item) {
    std::unique_lock<std::mutex> mlock(mutex_);
    while (queue_.empty()) {
      cond_.wait(mlock);
    }
    item = queue_.front();
    queue_.pop();
  }
  int push(const T& item) {
    int size;
    std::unique_lock<std::mutex> mlock(mutex_);
    queue_.push(item);
    size = queue_.size();

    if(size > 3)
        queue_.pop();
```

```
    mlock.unlock();
    cond_.notify_all();
    return size;
  }
 private:
  std::queue<T> queue_;
  std::mutex mutex_;
  std::condition_variable cond_;
};

#endif
```

Listing C.1: Concurrent message queue implementation.

Appendix D

Appendix D

D.1 Functions for trajectory estimation

```
void CTrajectory::update_list(double x, double y, double z) {
    Eigen::Vector3f temp;
    Eigen::Vector3f temporigin;
    double coords[3];

    for (int i = 1; i < HISTORY; i++) {
        previous_pose[i-1] = previous_pose[i];
        trajecto_pose[i-1] = trajecto_pose[i];
    }

    if (x != -1.0 && y != -1.0 && z != -1.0) {
        previous_pose[HISTORY-1][0] = x;
        previous_pose[HISTORY-1][1] = y;
        previous_pose[HISTORY-1][2] = z;

        if (has_trajectory) {
            temp = previous_pose[HISTORY-1];
            temp[2]=0;
            temporigin = origin;
            temporigin[2]=0;
            double d = (temp-temporigin).norm();
            compute_3D_from_1D(d,coords);

            if (sqrt((coords[0] - previous_pose[HISTORY-1][0]) * (coords[0] -
    previous_pose[HISTORY-1][0])
        + (coords[1] - previous_pose[HISTORY-1][1]) * (coords[1] -
    previous_pose[HISTORY-1][1])
```

```
        +(coords[2]-previous_pose[HISTORY-1][2]) * (coords[2]-previous_pose[
    HISTORY-1][2]) ) < MIN_DISTANCE_TRAJECTORY )
                trajecto_pose[HISTORY-1] = 1;
            else
                trajecto_pose[HISTORY-1] = 0;
        }
        else
            trajecto_pose[HISTORY-1] = 0;
    }
    else {
        previous_pose[HISTORY-1][0] = 0;
        previous_pose[HISTORY-1][1] = 0;
        previous_pose[HISTORY-1][2] = 0;
        trajecto_pose[HISTORY-1] = -1;
    }
}
```

Listing D.1: This function is in charge of receiving the new position of the ball and updating the information related to the trajectory in case the position is valid.

```
void CTrajectory::compute_trajectory_from_last_three() {
    Eigen::Vector3f temp;
    Eigen::Vector3f temporigin;
    double coords1[3],coords2[3],coords3[3];
    has_trajectory = 0;
    c(0)=0;
    origin(0) = origin(1)=origin(2)=0;

    if (trajecto_pose[HISTORY-3]>=0 && trajecto_pose[HISTORY-2]>=0 &&
    trajecto_pose[HISTORY-1]>=0) {
        double d1,d2,d3;
        origin = previous_pose[HISTORY-3];
        temporigin = origin;

        trajecto_pose[HISTORY-3] = 1;
        trajecto_pose[HISTORY-2] = 1;
        trajecto_pose[HISTORY-1] = 1;

        temporigin[2]=0;
        d3 = 0;
        temp = previous_pose[HISTORY-2];
        temp[2]=0;
        d2 = (temp-temporigin).norm();
        temp = previous_pose[HISTORY-1];
        temp[2]=0;
```

```
    d1 = (temp-temporigin).norm();

    Eigen::MatrixXf A(3,3);
    A <<  d3*d3,d3,1,d2*d2,d2,1,d1*d1,d1,1;
    Eigen::VectorXf b(3,1);
    b << previous_pose[HISTORY-3][2],previous_pose[HISTORY-2][2],
previous_pose[HISTORY-1][2];
    c = A.colPivHouseholderQr().solve(b);

    compute_3D_from_1D(d3,coords3);
    compute_3D_from_1D(d2,coords2);
    compute_3D_from_1D(d1,coords1);

    if ( sqrt ( (coords3[0]-previous_pose[HISTORY-3][0]) * (coords3[0]-
previous_pose[HISTORY-3][0])
                +(coords3[1]-previous_pose[HISTORY-3][1]) * (coords3[1]-
previous_pose[HISTORY-3][1])
                +(coords3[2]-previous_pose[HISTORY-3][2]) * (coords3[2]-
previous_pose[HISTORY-3][2]) ) > 10 * MIN_DISTANCE_TRAJECTORY
            || sqrt ( (coords2[0]-previous_pose[HISTORY-2][0]) * (coords2
[0]-previous_pose[HISTORY-2][0])
                +(coords2[1]-previous_pose[HISTORY-2][1]) * (coords2[1]-
previous_pose[HISTORY-2][1])
                +(coords2[2]-previous_pose[HISTORY-2][2]) * (coords2[2]-
previous_pose[HISTORY-2][2]) ) > 10 * MIN_DISTANCE_TRAJECTORY
            || sqrt ( (coords1[0]-previous_pose[HISTORY-1][0]) * (coords1
[0]-previous_pose[HISTORY-1][0])
                +(coords1[1]-previous_pose[HISTORY-1][1]) * (coords1[1]-
previous_pose[HISTORY-1][1])
                +(coords1[2]-previous_pose[HISTORY-1][2]) * (coords1[2]-
previous_pose[HISTORY-1][2]) ) > 10 * MIN_DISTANCE_TRAJECTORY) {
        has_trajectory = 0;
        c(0)=0;
        origin(0) = origin(1)=origin(2)=0;
        trajecto_pose[HISTORY-3] = 0;
        trajecto_pose[HISTORY-2] = 0;
        trajecto_pose[HISTORY-1] = 0;
    } else {
        has_trajectory = 1;
        trajecto_pose[HISTORY-3]=1;
        trajecto_pose[HISTORY-2]=1;
        trajecto_pose[HISTORY-1]=1;
        origin = previous_pose[HISTORY-3];
    }
```

```
    }
}
```

Listing D.2: Implementation of the trajectory estimation from the last three valid positions of the ball.

```
void CTrajectory::refine_trajectory() {
    Eigen::Vector3f temp;
    Eigen::Vector3f temporigin;

    Eigen::MatrixXf A(HISTORY,3);
    Eigen::VectorXf b(HISTORY,1);
    for(int k=0;k<HISTORY;k++) {
        if (trajecto_pose[k]==1) {
            double d;
            temp = previous_pose[k];
            temp[2]=0;
            temporigin = origin;
            temporigin[2]=0;
            d = (temp-temporigin).norm();
            A(k,0) = d*d;
            A(k,1) = d;
            A(k,2) = 1;
            b(k,0) = previous_pose[k][2];
        }
        else {
            A(k,0) = 0;
            A(k,1) = 0;
            A(k,2) = 1;
            b(k,0) = origin[2];
        }
    }
    c = A.colPivHouseholderQr().solve(b);
}
```

Listing D.3: Refines the trajectory if a valid trajectory already exists.